COMP202
Complexity of Algorithms
Sorting

[See Sections 2.4, 4.1, 4.3, and 4.5 in Goodrich and Tamassia.]
Learning outcomes

1. Understand the sorting problem, and its fundamental importance in algorithms.

2. Have a wide knowledge of different types of sorting algorithms available (such as heap sort, merge sort, and quick sort), as well as their algorithmic complexity (i.e. running time).

3. Comprehend the idea behind the divide-and-conquer methods of merge-sorting and quick sorting.

4. Be able to explain the mechanism behind the randomized quick sort algorithm, and why randomization is used.
**Sorting**

*Sorting problem:* Given a collection, $C$, of $n$ elements (and a total ordering) arrange the elements of $C$ into *non-decreasing* order, e.g.

<table>
<thead>
<tr>
<th>45</th>
<th>3</th>
<th>67</th>
<th>1</th>
<th>5</th>
<th>16</th>
<th>105</th>
<th>8</th>
</tr>
</thead>
</table>

| 1  | 3  | 5  | 8  | 16 | 45 | 67  | 105|


Sorting is a fundamental algorithmic problem in computer science.

Many algorithms perform sorting (as a subroutine) during their execution. Hence, efficient sorting methods are crucial to achieving good algorithmic performance.

We will investigate various methods that we can use to sort items. Why several methods?
We may not always require a fully sorted list (for example), so some methods might be more appropriate depending upon the exact task at hand.

Sorting algorithms might be directly adaptable to perform additional tasks and directly provide solutions in this fashion.
Priority Queues

A *Priority Queue* is a container of elements, each having an associated *key*.

*Keys* determine the *priority* used in picking elements to be removed.

A *priority Queue* (PQ) has these fundamental methods:

- `insertItem(k,e)`: insert element `e` having key `k` into PQ.
- `removeMin()`: remove minimum element.
- `minElement()`: return minimum element.
- `minKey()`: return key of minimum element.

(Of course, we could have priority queues that are based on maintaining the maximum elements.)
How can we use a priority queue to perform sorting on a set $C$? Do this in two phases:

- **First phase:** *Put* elements of $C$ into an initially empty priority queue, $P$, by a series of $n \text{ insertItem}$ operations.

- **Second phase:** *Extract* the elements from $P$ in *non-decreasing* order using a series of $n \text{ removeMin}$ operations.
PQ Sorting - Algorithm

PQ-SORT(C, P)

▷ Input: An n element sequence C and a priority queue P.
▷ Output: The sequence C sorted using the total order relation.

1 while C ≠ ∅
2   do
3     e ← C.REMOVEFIRST()
4     P.INSERTITEM(e, e)
5 while P ≠ ∅
6   do
7     e ← P.REMOVEMIN()
8     C.INSERTLAST(e)
Heap Data Structure

A heap is a realization of a Priority Queue that is efficient for both insertions and deletions.

A heap allows insertions and deletions to be performed in logarithmic time.

In a heap the elements and their keys are stored in an almost complete binary tree. Every level of the binary tree, except possibly the last one, will have the maximum number of children possible.
In a heap $T$, for every node $v$ (excluding the root) the key at $v$ is greater than (or equal to) the key stored at its parent.
PQ/Heap implementation

An efficient realization of a heap can be achieved using an array for storing the elements (i.e. the vector representation of a tree that we discussed earlier).

So a heap can be implemented with the following:

- **heap**: A (nearly complete) binary tree $T$ containing elements with keys satisfying the *heap-order property*, stored in an array.

- **last**: A *reference* to the *last used* node of $T$ in this array representation.

- **comp**: A *comparator* function that defines the *total order relation* on keys and which is used to maintain the *minimum* (or maximum) element at the *root* of $T$.
PQ/Heap implementation
Insertion in heaps

Inserting a new item into a heap begins by adding this element to the bottom of the tree in the position of the first unused, or empty, child.

Then, if necessary, this new element “bubbles” its way up the heap until the heap-order property is restored.
Up-heap bubbling (insertion)
Up-heap bubbling (insertion) (cont.)

(d)

(e)
Deletion in a heap consists of removing the minimum (or maximum) element (at the root) from the heap. Then the bottom, right-most element in the heap (the element at the end of the array that stores the heap) is moved to the root.

To restore the heap-order property, this item then “sinks” or bubbles down the heap.
Down-heap bubbling (removal of top element)
Down-heap bubbling (cont.)

(c)
Heap performance

Since a heap is an almost-complete binary tree, it stores \( n \) items in a tree of height \( O(\log n) \). Thus we have the following summary of running times for operations that can be performed on a heap.

<table>
<thead>
<tr>
<th>Operation</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>minElement, minKey</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insertItem</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>removeMin</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>
Theorem: The heap-sort algorithm sorts a sequence, $S$, of $n$ comparable items in $O(n \log n)$ time, where

- Bottom-up construction of heap with $n$ elements takes $O(n \log n)$ time, and
- Extraction of $n$ elements (in increasing order) from the heap takes $O(n \log n)$ time.
Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

▶ Divide: If the input size is small then solve the problem directly; otherwise, divide the input data into two or more disjoint subsets.

▶ Recur: Recursively solve the sub-problems associated with subsets.

▶ Conquer: Take the solutions to sub-problems and merge into a solution to the original problem.
Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

- **Divide**: If the input size is small then solve the problem directly; otherwise, divide the input data into two or more disjoint subsets.

  - *Divide*: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more *disjoint* subsets.
The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

- **Divide**: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more *disjoint* subsets.

- **Recur**: *Recursively* solve the sub-problems associated with subsets.
Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

▶ **Divide**: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more *disjoint* subsets.

▶ **Recur**: *Recursively* solve the sub-problems associated with subsets.

▶ **Conquer**: Take the solutions to sub-problems and *merge* into a solution to the original problem.
MergeSort

MergeSort is one way we can apply the divide-and-conquer method to perform sorting. The MergeSort method consists of the following steps:

- **Divide**: If input sequence $S$ has 0 or 1 elements, then return $S$; otherwise, split $S$ into two sequences $S_1$ and $S_2$, each containing about $\frac{1}{2}$ of the elements in $S$.
- **Recur**: Recursively sort $S_1$ and $S_2$.
- **Conquer**: Put the elements back into $S$ by merging the sorted sequences $S_1$ and $S_2$ into a single sorted sequence.
MergeSort - Illustration

1. Divide in half

Split list equally

2. Recur

S_1

2. Recur

S_2

3. Merge
MergeSort - Example
Theorem: *Merging* two sorted sequences $S_1$ and $S_2$ takes $O(n_1 + n_2)$ time, where $n_1$ is the size of $S_1$ and $n_2$ is the size of $S_2$. 

Idea: During every recursive step, dividing each sublist takes time at most $O(n)$ time. Merging all of the lists in each level takes at most $O(n)$ time as well.

How many times do we recurse? This is at most $O(\log n)$ times.

Hence this gives the $O(n \log n)$ running time for MergeSort.
MergeSort - Analysis

**Theorem:** Merging two sorted sequences $S_1$ and $S_2$ takes $O(n_1 + n_2)$ time, where $n_1$ is the size of $S_1$ and $n_2$ is the size of $S_2$.

**Theorem:** MergeSort runs in $O(n \log n)$ time in the worst (and average) case.
MergeSort - Analysis

**Theorem:** Merging two sorted sequences $S_1$ and $S_2$ takes $O(n_1 + n_2)$ time, where $n_1$ is the size of $S_1$ and $n_2$ is the size of $S_2$.

**Theorem:** MergeSort runs in $O(n \log n)$ time in the worst (and average) case.

**Idea:** During every recursive step, dividing each sublist takes time at most $O(n)$ time. Merging all of the lists in each level takes at most $O(n)$ time as well.

How many times do we recurse? This is at most $O(\log n)$ times. Hence this gives the $O(n \log n)$ running time for MergeSort.
MergeSort - Analysis

O(n) time per level

O(n) n

O(\log n) height

O(n) n/2 n/2

O(n) n/4 n/4 n/4 n/4

O(n) time per level

Total time: \(O(n \log n)\)
Counting inversions

This example is inspired by (if not directly related to) some of the “ranking systems” that are becoming more popular on some websites.
Counting inversions

This example is inspired by (if not directly related to) some of the “ranking systems” that are becoming more popular on some websites.

Suppose that you’ve rated a set of films or books (for example). In particular, you’ve rated \( n \) films by ranking them from your most favorite (ranked at 1) to least favorite (ranked at \( n \)).

In order to give a recommendation to you, this website wants to compare your ratings of these films with those of other people (for the same films) to see how similar they are.

How can you do this?
In other words, how can you compare your rankings

1 2 3 4 5 6 7 8 9 10

to another ranking

2 7 10 4 6 1 3 9 8 5?

Or even to another person’s rankings

8 9 10 1 3 4 2 5 6 7?

Which one of these is “closest” to your rankings?
Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.
One proposed way of measuring the similarity is to count the number of *inversions*.

Suppose that

\[ a_1, a_2, a_3, \ldots, a_n \]

denotes a permutation of the integers 1, 2, \ldots, n. The pair of integers \( i, j \) are said to form an inversion if \( i < j \), but \( a_i > a_j \). (We can generalize this idea to any sequence of distinct integers.)
Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.

Suppose that

\[ a_1, a_2, a_3, \ldots, a_n \]

denotes a permutation of the integers 1, 2, \ldots, \( n \). The pair of integers \( i, j \) are said to form an inversion if \( i < j \), but \( a_i > a_j \). (We can generalize this idea to any sequence of distinct integers.)

We will count the number of inversions to measure the similarity of someone’s rankings to yours.
Counting inversions (cont.)

For example, the permutation

\[ 1 \ 2 \ 4 \ 3 \]

contains one inversion (the 4 and the 3), while the permutation

\[ 1 \ 4 \ 3 \ 2 \]

has three (the 3, 4 pair, the 2, 3 and the 2, 4 pair).

In other words, to find the number of inversions, we count the pairs \( i \neq j \) that are \textit{out of order} in the permutation.
Counting inversions (cont.)

The number of inversions can range from 0, for the permutation

\[ 1 \ 2 \ 3 \ \ldots \ n, \]

up to \( \binom{n}{2} = \frac{n(n-1)}{2} \) for the permutation

\[ n \ n-1 \ \ldots \ 2 \ 1. \]

Other examples:

\[ 2 \ 1 \ 3 \ 4 \ 5 \] has one inversion,
\[ 2 \ 3 \ 4 \ 5 \ 1 \] has four inversions,
\[ 5 \ 2 \ 3 \ 4 \ 1 \] has seven inversions, and
\[ 5 \ 4 \ 3 \ 1 \ 2 \] has nine inversions.
Counting inversions (cont.)

In terms of the ranking system describe earlier, the number of inversions for a permutation is a measure of how “out of order” it is as compared to the identity permutation

\[ 1 \ 2 \ 3 \ \ldots \ n \]

and hence could be used to measure the “similarity” to the identity permutation.
Counting inversions (cont.)

In terms of the ranking system describe earlier, the number of inversions for a permutation is a measure of how “out of order” it is as compared to the identity permutation

\[ 1 \ 2 \ 3 \ \ldots \ n \]

and hence could be used to measure the “similarity” to the identity permutation.

Note, however, that two different permutations can have the same number of inversions.

2 1 3 4 5

and

1 2 4 3 5

each have a single inversion, but are obviously not the same permutation.
Counting inversions: How do we do it?

So how do we count the number of inversions in a given permutation of \( n \) numbers?

The “naive” approach is to check all \( \binom{n}{2} \) pairs to see if they form an inversion in the permutation. This gives an algorithm with \( \Omega(n^2) \) running time.

Can we do better?
Counting inversions: How do we do it?

So how do we count the number of inversions in a given permutation of $n$ numbers?

The “naive” approach is to check all $\binom{n}{2}$ pairs to see if they form an inversion in the permutation. This gives an algorithm with $\Omega(n^2)$ running time.

Can we do better?

**Claim:** We can count inversions using a divide-and-conquer algorithm that runs in time $O(n \log n)$. 
A divide-and-conquer way to count inversions

Idea:
As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.
A divide-and-conquer way to count inversions

**Idea:**
As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.

To do that we can *sort* each sublist and merge them into a single (sorted) list. As we merge them together into a single list, we can count the inversions from such pairs mentioned above.
A divide-and-conquer way to count inversions

Idea:
As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.

To do that we can sort each sublist and merge them into a single (sorted) list. As we merge them together into a single list, we can count the inversions from such pairs mentioned above.

In other words, we’re performing a modified MergeSort!
Divide-and-conquer for counting inversions

Suppose that we’ve divided the list into $A$ (the first half) and $B$ (the second half) and have counted the inversions in each. After sorting them, the idea for counting the additional inversions is as follows:

Elements inverted with $b_j < a_i$

As we merge the lists, every time we take an element from the list $B$, it forms an inversion with all of the remaining (unused) elements in list $A$. 
A recursive algorithm for counting inversions

\texttt{COUNTINVERSIONS}(L)

- Input: A list, \( L \), of distinct integers.
- Output: The number of inversions in \( L \).

1. \textbf{if} \( L \) has one element in it \textbf{then}
2. \hspace{1em} there are no inversions, so Return \((0, L)\)
3. \textbf{else}
   - \textbf{Divide the list into two halves}
4. \hspace{1em} \( A \) contains the first \( \lfloor n/2 \rfloor \) elements
5. \hspace{1em} \( B \) contains the last \( \lceil n/2 \rceil \) elements
6. \hspace{1em} \((k_A, A) = \text{COUNTINVERSIONS}(A)\)
7. \hspace{1em} \((k_B, B) = \text{COUNTINVERSIONS}(B)\)
8. \hspace{1em} \((k, L) = \text{MERGEANDCOUNT}(A, B)\)
9. Return \((k_A + k_B + k, L)\)
The **MERGEANDCOUNT** method

**MERGEANDCOUNT**(*A*, *B*)

1. \( \text{Current}_A \leftarrow 0 \)
2. \( \text{Current}_B \leftarrow 0 \)
3. \( \text{Count} \leftarrow 0 \)
4. \( L \leftarrow \text{empty list} \)
5. **while** both lists (*A* and *B*) are non-empty
6. \hspace{1em} Let \( a_i \) and \( b_j \) denote the elements pointed to by \( \text{Current}_A \) and \( \text{Current}_B \).
7. \hspace{1em} Append the smaller of \( a_i \) and \( b_j \) to \( L \).
8. \hspace{1em} **if** \( b_j \) is the smaller element **then**
9. \hspace{2em} Increase \( \text{Count} \) by the number of elements remaining in \( A \).
10. \hspace{1em} Advance the \( \text{Current} \) pointer of the appropriate list.
11. Once one of \( A \) and \( B \) is empty, append the remaining elements to \( L \).
12. Return \((\text{Count}, L)\)
Counting inversions - The payoff

As mentioned earlier, this method for counting inversions is basically a modified version of the MergeSort algorithm.

Hence, we can count the number of inversions in a permutation in time $O(n \log n)$. 
The QuickSort algorithm is yet another divide-and-conquer method for performing sorting. This method differs from MergeSort in that the steps for QuickSort are as follows:

- **Divide**: If $|S| > 1$, select a *pivot element* $x$ in $S$ and create three sequences: $L$, $E$, and $G$ where
  - $L$ stores elements in $S$ that are less than $x$,
  - $E$ stores elements in $S$ that are equal to $x$, and
  - $G$ stores elements in $S$ that are greater than $x$.

- **Recur**: Recursively sort the sequences $L$ and $G$.

- **Conquer**: Put sorted elements from $L$, $E$, and $G$ together to form a sorted list.
QuickSort Tree

1. Split using pivot $x$

2. Recur

3. Concatenate

2. Recur

$L < x$

$E = x$

$R > x$
QuickSort - Example
QuickSort

What is the running time of the QuickSort algorithm?

- Let $s_i$ be the sizes of the input nodes (at depth $i$) in the tree.
- $s_i \leq n - i$. (Note that $s_i = n - i$, when use of pivots leads to a single empty sequence (for $L$ or $G$).)
- The worst case complexity is bounded by

\[
O\left(\sum_{i=0}^{n-1} s_i\right) = O\left(\sum_{i=0}^{n-1} n - i\right) = O\left(\sum_{i=0}^{n-1} i\right).
\]

Since

\[
\sum_{i=0}^{n-1} i = \frac{(n - 1)n}{2}
\]

the worst case running time of QuickSort is $O(n^2)$. 
Randomized QuickSort

The worst case running time of QuickSort occurs when one of the sublists is in fact empty.

To avoid this worst case run time, we want to ensure that each of $L$ and $G$ are sufficiently large for each recursive call.

For any *deterministic* method of choosing a pivot element, we can design some list so that the running time is (almost) $n^2$. (And we can design a set of lists so that QuickSort takes, say, at least $n^{3/2}$.)

Solution: Select a pivot element randomly!
Randomized QuickSort

The worst case running time of QuickSort occurs when one of the sublists is in fact empty.

To avoid this worst case run time, we want to ensure that each of $L$ and $G$ are sufficiently large for each recursive call.

For any deterministic method of choosing a pivot element, we can design some list so that the running time is (almost) $n^2$. (And we can design a set of lists so that QuickSort takes, say, at least $n^{\frac{3}{2}}$.)

Solution: Select a pivot element randomly!
Basic Probability

- Probability theory is useful in many contexts in the theory of algorithms.

It can be used to analyze the *average-case* performance of algorithms (by *weighting* the running time according to a probability distribution on inputs.)
Basic Probability

- Probability theory is useful in many contexts in the theory of algorithms.

It can be used to analyze the *average-case* performance of algorithms (by *weighting* the running time according to a probability distribution on inputs.

*Randomized algorithms* (those that use random bits during their execution to make decisions) are also becoming more commonplace and useful in recent years. Analysis of such randomized algorithms also requires use of probability theory.

See Section 1.3.4 in [GT] for some details on probability theory.
Sample space

Sample space is the set of all possible outcomes from an experiment.

Examples:

1. Rolling a fair (6-sided) die. The sample space is \{1, 2, 3, 4, 5, 6\}.

2. Flipping a fair coin. The sample space is \{H, T\}.

3. Flipping a fair coin until it comes up heads. Here the sample space is infinite, with the \(i^{th}\) possible outcome being a sequence of \(i - 1\) tails followed by a single heads, for \(i = 1, 2, 3, 4, \ldots\)
A *Probability space* is a sample space \( S \) and a *probability function* \( Pr \), which maps subsets of \( S \) to \([0, 1]\).

Each subset, \( A \subseteq S \), is called an *event*. 
A probability function, $Pr$, possesses the following properties:

1. $Pr(\emptyset) = 0$,
2. $Pr(S) = 1$,
3. $0 \leq Pr(A) \leq 1$, for any $A \subseteq S$,
4. If $A, B \subseteq S$ and $A \cap B = \emptyset$, then

$$Pr(A \cup B) = Pr(A) + Pr(B).$$
An example of a probability space

Consider the sample space earlier, namely that of flipping a fair coin until it comes up heads. We can represent the sample space as an infinite set of strings, namely \( \{H, TH, TTH, TTTTH, \ldots \} \).
An example of a probability space

Consider the sample space earlier, namely that of flipping a fair coin until it comes up heads. We can represent the sample space as an infinite set of strings, namely \{H, TH, TTH, TTTH, \ldots\}. The probability function on this sample space assigns these probabilities to events:

\[
\begin{align*}
Pr(H) & = 1/2 \\
Pr(TH) & = 1/4 \\
Pr(TTH) & = 1/8 \\
& \ldots \\
Pr(H \text{ or } TTH) & = 1/2 + 1/8 = 5/8 \\
& \ldots
\end{align*}
\]
An example of a probability space

Consider the sample space earlier, namely that of flipping a fair coin until it comes up heads. We can represent the sample space as an infinite set of strings, namely \{H, TH, TTH, TTTH, \ldots\}. The probability function on this sample space assigns these probabilities to events:

\[
\begin{align*}
Pr(H) & = 1/2 \\
Pr(TH) & = 1/4 \\
Pr(TTH) & = 1/8 \\
\vdots \\
Pr(H \text{ or } TTH) & = 1/2 + 1/8 = 5/8 \\
\vdots
\end{align*}
\]

(These statements make use of the independence of coin flips discussed on the next slide, and the last statement uses the final property of the probability function on the previous slide.)
Independent Events

- Two events are *independent* if \( \Pr(\mathcal{A} \cap \mathcal{B}) = \Pr(\mathcal{A}) \cdot \Pr(\mathcal{B}) \).
- A collection of events \( \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\} \) is *mutually independent* if

\[
\Pr(\mathcal{A}_{i_1} \cap \mathcal{A}_{i_2} \cap \ldots \cap \mathcal{A}_{i_k}) = \Pr(\mathcal{A}_{i_1}) \cdot \Pr(\mathcal{A}_{i_2}) \cdot \ldots \cdot \Pr(\mathcal{A}_{i_k})
\]

for any subcollection of events \( \{\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_k}\} \).
Example

Let $A$ be the event that the roll of a fair die is 6, and let $B$ be the event that the roll of a second fair die is a 3, and let $C$ be the event that the sum of the two dice is 10.

Then $A$ and $B$ are independent events, but it’s easy to check that $C$ is not independent with either $A$ or $B$.

This follows since

$$ Pr(A) = Pr(B) = \frac{1}{6} $$

$$ Pr(C) = \frac{3}{36} $$

$$ Pr(A \cap C) = \frac{1}{36} = Pr(A \cap B), \text{ and} $$

$$ Pr(B \cap C) = 0. $$
Random variables

- A random variable is a function $X$ that maps outcomes from some sample space to real numbers.

In algorithm analysis we often use a random variable $X$ with a discrete set of outcomes to characterize the running time of a randomized algorithm.
A *random variable* is a function $X$ that maps outcomes from some sample space to real numbers.

In algorithm analysis we often use a random variable $X$ with a discrete set of outcomes to characterize the running time of a randomized algorithm.

The *expected value* of a discrete random variable $X$ is defined as $\mathbb{E}(X) = \sum_i i \cdot Pr(X = i)$ where the summation is over the range of possible values of $X$.

The expected value is *linear*, meaning that for any two random variables $X$ and $Y$ we have $\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$. 
Example

Consider the event of rolling a fair die, and let $X$ denote the value that occurs. Then we have

$$Pr(X = 1) = Pr(X = 2) = \ldots = Pr(X = 6) = \frac{1}{6}.$$ 

By definition, we have

$$\mathbb{E}(X) = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3 \frac{1}{2}.$$
Example (cont.)

Suppose that $Y$ is the value obtained by the roll of another fair die. Then $X + Y$ represents the sum of the two dice. We have

$$
\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y) = 3\frac{1}{2} + 3\frac{1}{2} = 7,
$$

which can also be computed from the definition of expected value using the facts that

$$
\begin{align*}
Pr(X + Y = 2) &= 1/36 \\
Pr(X + Y = 3) &= 2/36 \\
&\quad\vdots \\
Pr(X + Y = 6) &= 5/36 \\
Pr(X + Y = 7) &= 6/36 \\
Pr(X + Y = 8) &= 5/36 \\
&\quad\vdots
\end{align*}
$$
Another example

Consider the problem from before of flipping a fair coin until it comes up heads for the first time. Let $X$ denote the number of flips that must be made until the heads appears (including the final flip). Then $X$ is a random variable that takes values in $\{1, 2, 3, \ldots\}$. We note that

$$\Pr(X = i) = \left(\frac{1}{2}\right)^i \text{ for } i \in \{1, 2, 3, \ldots\}.$$ 

By definition, we then have

$$\mathbb{E}(X) = \sum_{i=1}^{\infty} i \cdot Pr(X = i) = \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i.$$ 

Using a bit of calculus (which I don’t necessarily expect you to know), we can find that $\mathbb{E}(X) = 2$. 
Another way to get the same result

We can get the same answer for the preceding example in another way. If you consider the first coin toss, either it comes up heads (in which case the experiment is over), or it doesn’t (which “costs” us one coin toss, plus however many more we need until the first heads shows up).

Thus we can write

$$\mathbb{E}(X) = Pr(X \text{ is heads}) \cdot 1 + Pr(X \text{ is tails}) \cdot (1 + \text{expected number of additional coin tosses until first heads appears}).$$

But the “expected number of additional coin tosses until the first heads appears” is exactly the same as $\mathbb{E}(X)!$ (Having first obtained a “tails” the experiment “resets itself” and we’re just interested in obtaining the first heads again.)
Thus we have

$$E(X) = Pr(X \text{ is heads}) \cdot 1 + Pr(X \text{ is tails}) \cdot (1 + E(X)).$$

Of course,

$$Pr(X \text{ is heads}) = Pr(X \text{ is tails}) = \frac{1}{2},$$

so

$$E(X) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + E(X))$$

which upon solving for $E(X)$ gives us the same result, namely $E(X) = 2$. 
Recall that our proposed method for the randomized QuickSort algorithm is to choose the pivot element at random. We can repeat the selection until we find a “good” pivot element (defined below). With a good pivot element during each step, we have an algorithm with (expected) running time that is $o(n^2)$. 

**Theorem**

The expected running time of randomized QuickSort (pivot chosen at random) is $O(n \log n)$.

**Proof idea:**

We say that a random pivot is good if the size of neither $L$ nor $G$ is larger than $3/4 |S|$. 


Recall that our proposed method for the randomized QuickSort algorithm is to choose the pivot element at random.

We can repeat the selection until we find a “good” pivot element (defined below). With a good pivot element during each step, we have an algorithm with (expected) running time that is $o(n^2)$.

**Theorem** The expected running time of randomized QuickSort (pivot chosen at random) is $O(n \log n)$. 
Randomized QuickSort (cont.)

Recall that our proposed method for the randomized QuickSort algorithm is to choose the pivot element at random.

We can repeat the selection until we find a “good” pivot element (defined below). With a good pivot element during each step, we have an algorithm with (expected) running time that is $o(n^2)$.

**Theorem** The expected running time of randomized QuickSort (pivot chosen at random) is $O(n \log n)$.

*Proof idea:*

- We say that a random pivot is *good* if the size of neither $L$ or $G$ is larger than $\frac{3}{4} |S|$. 
Randomized QuickSort (cont.)

- The probability of success of choosing a good pivot is $\frac{1}{2}$.

- Fact: The *expected* number of times that a *fair* coin must be tossed until it shows heads $k$ times is $2^k$. Choosing a good pivot is analogous to flipping a fair coin until it comes up heads. So, on average we need two attempts (random choices) to find a good pivot.
Randomized QuickSort (cont.)

- The probability of success of choosing a good pivot is $\frac{1}{2}$.

- Fact: The *expected* number of times that a *fair* coin must be tossed until it shows heads $k$ times is $2^k$. Choosing a good pivot is analogous to flipping a fair coin until it comes up heads. So, on average we need two attempts (random choices) to find a good pivot.

- Any path in the QuickSort algorithm can contain at most $\log_{4/3} n$ levels with good pivots.

- Hence, the expected length of each path will be $2 \log_{4/3} n$. 
Randomized QuickSort

Expected height: $O(\log n)$

Total running time: $O(n \log n)$
Bucket Sort is not based on comparisons but on using keys as indices of a bucket array $B$ that has entries within an integer range $[0, 1, \ldots, N - 1]$.

Initially all items from input sequence, $S$, are moved to appropriate buckets, i.e., an item with key $k$ is moved to bucket $B[k]$.

Then move all items back into $S$ according to their order of appearance in consecutive buckets $B[0], B[1], \ldots, B[N - 1]$.
Bucket Sorting: An example

Bucket sorting (or some variant) is performed by the postal service to deliver mail. Letters and parcels are sorted based on postal addresses, then gathered up for the actual delivery process.

(Typically this could actually require several bucket sorts, e.g. first on county, then on town, then on post code, etc.)
Bucket Sort - Algorithm

**BUCKETSORT**(S)

- Input: A sequence S of items with integer keys in range \([0, 1, \ldots, N - 1]\).
- Output: Sequence S sorted in non-decreasing order of keys.

1. for each item x in S
2. do
3.   ▶ Let k be key of item x
4.   \( B[k] \leftarrow REMOVE(x, S) \)
5. for i ← 0 to \( N - 1 \)
6.   do
7.     for each item x in sequence \( B[i] \)
8.     do
9.       \( S \leftarrow REMOVE(x, B[i]) \)
Radix Sorting

Radix sorting was a method that was used by punched card sorting machines.

It is based on a card sorting technique which sorts column by column using a bucket sorting algorithm on each successive column.
Radix sorting can be used to sort ordered pairs into *lexicographic* order (i.e. “dictionary” order), where each coordinate lies in the range of integers $[1, 2, \ldots, N]$ (or perhaps $[0, 1, \ldots, N-1]$).

Lexicographic ordering is defined so that if $(k_1, \ell_1)$ and $(k_2, \ell_2)$ are ordered pairs, then we say that $(k_1, \ell_1) < (k_2, \ell_2)$ if

$$k_1 < k_2, \text{ or } k_1 = k_2 \text{ and } \ell_1 < \ell_2.$$
In general, we can use radix sorting to sort ordered $d$-tuples into lexicographic ordering (defined analogously as above for more than two coordinates).

Radix sorting can also be used to sort integers of a fixed length $d$ (or integers with \textit{at most} $d$ digits).

In the punched card case, each card has $d$ columns (in practice $d = 12$ was common).
Radix Sorting (cont.)

- The mechanism of radix sorting is to *stably* sort on each of $d$ columns successively (starting with the least significant column).

A sorting algorithm is said to be *stable* if it is *order-preserving*, i.e. if there is a tie on $i^{th}$ column, then the original order of the two elements is unchanged by the sorting algorithm.
Radix Sorting (cont.)

- Sorting from the *least significant* to *most significant* digit (or from right to left for ordered $d$-tuples) is vital, and, in fact, is what makes the algorithm work (the other way around won’t necessarily give the correct result).

Radix sort makes $d$ passes over the data, and sorts on a *single* column on each pass.

In iteration $i$ a stable sorting algorithm is used to sort column $i$ (i.e. the $i$th column from the right).
Radix Sort - Algorithm

RadixSort(S, d)

▷ Input: Array S and integer d.
▷ Output: Array S with elements in increasing order.

1  \textbf{for} \ i \leftarrow 1 \ \textbf{to} \ d \\
2 \quad \textbf{do} \\
3 \quad \text{Stably bucket-sort S on } i\text{th column from the right.}
### Radix Sorting - An example

<table>
<thead>
<tr>
<th></th>
<th>d=1</th>
<th></th>
<th>d=2</th>
<th></th>
<th>d=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>159</td>
<td>742</td>
<td>108</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>496</td>
<td>125</td>
<td>125</td>
<td>125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>357</td>
<td>496</td>
<td>742</td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>896</td>
<td>357</td>
<td>357</td>
<td></td>
<td></td>
</tr>
<tr>
<td>896</td>
<td>357</td>
<td>159</td>
<td>496</td>
<td></td>
<td></td>
</tr>
<tr>
<td>742</td>
<td>108</td>
<td>496</td>
<td>742</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>159</td>
<td>896</td>
<td>896</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Radix sorting

It is essential that the digit sort be stable because the order preservation (in the case of ties) is *extremely* important.

Radix sort can be used to sort a set of records that may be keyed on multiple fields.

Another example is sorting by date, provided we do it in the correct order (day first, then month, and finally year).

Radix sort is a straightforward algorithm that can give us improved efficiency in some cases (when compared to, for example, *bubble* sort).

We can also use radix sort for *binary* sequences.
Radix Sorting - Analysis

Suppose that we have \( n \) items to sort, each of which is a \( d \)-tuple (or integer with at most \( d \) digits), and each coordinate lies in the range \([0, 1, \ldots, N−1]\).

Then, radix sorting will sort this sequence in time \( O(d(n + N)) \).

For example, sorting integers with (at most) \( d \) digits, we have that \( N = 10 \) (since each digit is one of \([0, 1, \ldots, 9]\)), so the running time to sort the integers is in \( O(d(n + 10)) \), i.e. \( O(dn) \).

We only beat the \( O(n \log n) \) time of MergeSort or randomized QuickSort because of the fixed length of the integers.