CS 2413: Data Structures (Fall’20)
Search Algorithms and Big O Time Complexity

Chao Lan
Additional Reference
Outline

What is search?

Linear Search Algorithm: examples of success and failure.

Binary Search Algorithm: examples of success and failure, other issues.

Recursive Algorithm: binary search (pseudo-code and code), Fibonacci sequence

Time Complexity and Big O
What is a search problem?

Given a list of elements, search for a target element (which may or may not be in the list).

Example: search for element 9 in the following list; return its position or return failure.

| 4 | 7 | 1 | 12 | 9 | 3 | 10 |

The target element is also called a search key.
What is a search problem?

A list often has its N elements indexed, with a **head** (index: 0) and a **tail** (index: N-1).

* This is a virtual list, not necessary an array in C++. 

```
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
```

↑  head

↑  tail
Linear Search Algorithm

Linear search algorithm aims to go over all elements in the list, from head to tail, one by one, until the search key is found (stop search; return index) or not found (return failure, e.g., -1).

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
Let’s see an input scenario where linear search succeeds!
A scenario where linear search succeeds.

Linear search algorithm aims to go over all elements in the list, from head to tail, one by one, until the search key is found (stop search; return index) or not found (return failure, e.g., -1).

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Check the 1st element.

It is not 9!

Recall 9 is search key.
A scenario where linear search succeeds.

Linear search algorithm aims to go over all elements in the list, from head to tail, one by one, until the search key is found (stop search; return index) or not found (return failure, e.g., -1).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Check the 2nd element.

It is not 9!
A scenario where linear search succeeds.

Linear search algorithm aims to go over all elements in the list, from head to tail, one by one, until the search key is found (stop search; return index) or not found (return failure, e.g., -1).

Check the 3rd element.

It is not 9!
A scenario where linear search succeeds.

Linear search algorithm aims to go over all elements in the list, from head to tail, one by one, until the search key is found (stop search; return index) or not found (return failure, e.g., -1).

Check the 4th element.

It is not 9!
A scenario where linear search succeeds.

Linear search algorithm aims to go over all elements in the list, from head to tail, one by one, until the search key is found (stop search; return index) or not found (return failure, e.g., -1).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Check the 5th element.

It is 9!

Stop search and return index 4.
Let’s see another input scenario where linear search fails!
Another scenario where linear search fails.

Suppose search key is 6.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Check the 1st element.

It is not 6.
Another scenario where linear search fails.

Suppose search key is 6.

Check the 2nd element.

It is not 6.
Another scenario where linear search fails.

Suppose search key is 6.

Check the 3rd element.

It is not 6.
Another scenario where linear search fails.

Suppose search key is 6.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Check the 4th element.

It is not 6.
Another scenario where linear search fails.

Suppose search key is 6.

Check the 5th element.
It is not 6.
Another scenario where linear search fails.

Suppose search key is 6.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Check the 6th element. It is not 6.
Another scenario where linear search fails.

Search fails because our algorithm does not find 6 in the list!

```
  0  1  2  3  4  5  6
  4  7  1 12  9  3 10
```

Check the last element.

Stop search and return failure info.

e.g., return -1.

It is not 6.
Outline

What is search?

Linear Search Algorithm: examples of success and failure.

Binary Search Algorithm: examples of success and failure, other issues.

Recursive Algorithm: binary search (pseudo-code and code), Fibonacci sequence

Time Complexity and Big O
Binary Search Algorithm

1. Assume elements are sorted, e.g., in an ascending order.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

2. Recursively
   - check midpoint of the current search domain
   - halve current domain if midpoint is not key, or return midpoint index if it is the key
   - pick one subdomain and treat it as the new domain for the next round of search.

* We will focus discussions on the ascending order. Discussions on descending order are similar.
Example binary search process. (key = 4)

This is our initial search domain.

Round 1

Check midpoint of the current domain. It is not 4!
Example binary search process. (key = 4)

Based on midpoint, halve the current domain into a left subdomain and a right subdomain.
Example binary search process. (key = 4)

Search key 4 is smaller than midpoint 7, so narrow domain to the left (for next round of search).

-- do you know why left instead of right?
### Example binary search process. (key = 4)

This is our current search domain.

| 1 | 3 | 4 | 7 | 9 | 10 | 12 |

Round 2

Check midpoint of the current domain. It is not 4!
Example binary search process. (key = 4)

Round 2

Based on midpoint, halve the current domain into a left subdomain and a right subdomain.
Example binary search process. (key = 4)

Search key 4 is bigger than midpoint 3, so narrow domain to the right (for next round of search).

--- do you know why right instead of left?

End of round 2 search.
Example binary search process. (key = 4)

This is our current search domain.

Round 3

1 3 4 7 9 10 12

Check midpoint of the current domain. It is 4!

Return index 2 and terminate search!
Let’s see another scenario where binary search fails!
Example binary search process. \((key = 8)\)

This is our initial search domain.

Round 1

Check midpoint of the current domain. It is not 8!
Example binary search process. (key = 8)

Based on midpoint, halve the current domain into a left subdomain and a right subdomain.
Example binary search process. (key = 8)

Search key 8 is bigger than midpoint 7, so narrow domain to the right (for next round of search).
Example binary search process. (key = 8)

This is our current search domain. **Round 2**

Check midpoint of the current domain. It is not 8!
Example binary search process. (key = 8)

Based on midpoint, halve the current domain into a left subdomain and a right subdomain.
Example binary search process. (key = 8)

Search key 8 is smaller than midpoint 10, so narrow domain to the left (for next round of search).
Example binary search process. (key = 8)

| 1 | 3 | 4 | 7 | 9 | 10 | 12 |

This is our current search domain.

Round 3

Check midpoint of the current domain. It is not 8!

But we have no element left to examine.

So stop search and return failure.
We have two more questions on binary search practice!
Q1: to the left, or to the right? (key = 2)

It is not 2!

pick left or right for next round of search?
Q2: what if we have an even number of inputs?

If there is an odd number of inputs, we can pinpoint the **exact midpoint**.

| 1 | 3 | 4 | 7 | 9 | 10 | 12 |

If there is an even number of inputs...

| 1 | 3 | 4 | 7 | 8 | 9 | 10 | 12 |
Q2: what if we have an even number of inputs?

If there is an odd number of inputs, we can pinpoint the exact midpoint.

| 1 | 3 | 4 | 7 | 9 | 10 | 12 |

If there is an even number of inputs, we can randomly pick one nearest to the midpoint.

| 1 | 3 | 4 | 7 | 8 | 9 | 10 | 12 |

Either 7 or 8 will do!
Outline

What is search?

Linear Search Algorithm: examples of success and failure.

Binary Search Algorithm: examples of success and failure, other issues.

Recursive Algorithm: binary search (pseudo-code and code), Fibonacci sequence

Time Complexity and Big O
Some algorithms are recursive by nature, e.g.,
binary search recursively checks midpoint and picks one subdomain (if needed).
Recursive Algorithm

Some algorithms are recursive by nature, e.g.,
binary search recursively checks midpoint and picks one subdomain (if needed).

Check midpoint and pick a subdomain.
Some algorithms are recursive by nature, e.g.,
binary search recursively checks midpoint and picks one subdomain (if needed).

Check midpoint and pick a subdomain.
Efficient implementation of recursive algorithm.

Recursive algorithm can be efficiently implemented using recursive function that calls itself.

```
BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}
```

Define a recursive function BinarySearch.

Input: search domain and key.

Output: index of the key, or -1 if fail.

Note this function calls itself!

We will first go over pseudo-code and later address some technical details such as how to describe the domain.
Let's go over an input scenario of the pseudo-code where binary search succeeds!
Example Recursion Process

Search Key = 8.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4, 5, 7, 8, 9\}.

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Call function with current domain and key.
Example Recursion Process

Round 1

BinarySearch (search domain + key) {

  if domain is invalid, stop search

  examine middle element

  if success, stop and return index;

  if failed, narrow search domain

  BinarySearch(search domain + key)

}
Example Recursion Process

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}
Example Recursion Process

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

Fail. so narrow search domain to {7, 8, 9}.
Example Recursion Process

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Call function itself, with an updated domain!
Recursion!
Example Recursion Process

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

Round 2

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Call function with current domain and key.
Example Recursion Process

Round 2

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

Current domain is valid.
Example Recursion Process

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Round 2

Examine 8, it is the search key!
Example Recursion Process

Round 2

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

Success! Stop, return index 5 (initial domain).
Example Recursion Process

Search Key = 8.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4, 5, 7, 8, 9\}.

Round 1

BinarySearch ( ) { ... }
Example Recursion Process

Search Key = 8.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{7, 8, 9\}.

Round 1  \[\text{BinarySearch ( ) \{ ... \}}\]

\[\downarrow\]

Round 2  \[\text{BinarySearch ( ) \{ ... \}}\]
Example Recursion Process

Search Key = 8.

Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.

Current Domain = \{7, 8, 9\}.

Round 1

BinarySearch ( ) { … }

Round 2

BinarySearch ( ) { … }

When we run BinarySearch in Round 2, the BinarySearch in Round 1 remains open.
Example Recursion Process

Search Key = 8.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {7, 8, 9}.

Round 1
BinarySearch ( ) { … }

Round 2
BinarySearch ( ) { … }
Find key. Return index 6.
Let’s go over another input scenario of the pseudo-code where binary search fails!
Example Recursion Process

Round 1

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Call function with current domain and key.

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.  
Current Domain = \{1, 3, 4, 5, 7, 8, 9\}.  

Example Recursion Process

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1, 3, 4, 5, 7, 8, 9}.

Current domain is valid.
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4, 5, 7, 8, 9\}.

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Examine 5, not search key.
Example Recursion Process

Round 1

```
BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}
```

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1, 3, 4}.

Fail. so narrow search domain to {1, 3, 4}.
Example Recursion Process

Round 1

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1, 3, 4}.

Call function itself, with an updated domain!

Recursion!
Example Recursion Process

Round 2

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Call function with current domain and key.

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4\}. 

Example Recursion Process

Round 2

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4\}.
Examine 3, not search key.

Example Recursion Process

Round 2

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1, 3, 4}.
Example Recursion Process

Round 2

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}
Example Recursion Process

Round 2

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1}.

Call function itself, with an updated domain!

Recursion!
Example Recursion Process

Round 3

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch (search domain + key)
}

Call function with current domain and key.

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1}. 
Example Recursion Process

Round 3

BinarySearch (search domain + key) {
  if domain is invalid, stop search
  examine middle element
  if success, stop and return index;
  if failed, narrow search domain
  BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1}.
Example Recursion Process

Round 3

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1\}.

Examine 1, not search key.

Halve \{1\} \approx \{\}, 1, \{\}.

Both subdomains are empty.
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{\}.

Round 3

```
BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}
```

Fail. so narrow search domain to \{\}. (empty)
Example Recursion Process

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}. Current Domain = { }.

Round 3

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Call function itself, with an updated domain!
Recursion!
Example Recursion Process

Round 4

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Call function with current domain and key.

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = { }.
Example Recursion Process

Round 4

BinarySearch (search domain + key) {
    if domain is invalid, stop search
    examine middle element
    if success, stop and return index;
    if failed, narrow search domain
    BinarySearch(search domain + key)
}

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = { }.  

Current domain is invalid!
Stop and return failure (e.g., return index -1).
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4, 5, 7, 8, 9\}.

Round 1          BinarySearch ( ) { … }
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{1, 3, 4\}.

Round 1

BinarySearch ( ) \{ ... \}

Round 2

BinarySearch ( ) \{ ... \}
Example Recursion Process

Search Key = 2.
Initial Domain = {1, 3, 4, 5, 7, 8, 9}.
Current Domain = {1}.

Round 1
BinarySearch ( ) { … }

Round 2
BinarySearch ( ) { … }

Round 3
BinarySearch ( ) { … }
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{\}\n
Round 1
\[
\text{BinarySearch ( )} \{ \ldots \}
\]

Round 2
\[
\text{BinarySearch ( )} \{ \ldots \}
\]

Round 3
\[
\text{BinarySearch ( )} \{ \ldots \}
\]

Round 4
\[
\text{BinarySearch ( )} \{ \ldots \}
\]
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{ \}.

In round 4, all previous BinarySearch calls are open.

Round 1
\text{BinarySearch} ( ) \{ … \}

Round 2
\text{BinarySearch} ( ) \{ … \}

Round 3
\text{BinarySearch} ( ) \{ … \}

Round 4
\text{BinarySearch} ( ) \{ … \}
Example Recursion Process

Search Key = 2.
Initial Domain = \{1, 3, 4, 5, 7, 8, 9\}.
Current Domain = \{\}\.

Round 1
\texttt{BinarySearch ( ) \{ ... \}}

Round 2
\texttt{BinarySearch ( ) \{ ... \}}

Round 3
\texttt{BinarySearch ( ) \{ ... \}}

Round 4
search fails, return -1. \texttt{ BinarySearch ( ) \{ ... \}}
Outline

What is search?

Linear Search Algorithm: examples of success and failure.

Binary Search Algorithm: examples of success and failure, other issues.

Recursive Algorithm: binary search (pseudo-code and code), Fibonacci sequence

Time Complexity and Big O
Assume initial list is stored in an array and pass to function via pointer.
Example code of recursive binary search function.

```c
int BinarySearch(int* x, int key, int head, int end){
    if (head > end) {return -1;}
    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    } else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

It is a recursive function because it calls itself.

It is very compact!
Characterize search domain using a “head” and “end” indices.

```c
int BinarySearch(int* x, int key, int head, int end) {
    // head: index of head in the current domain!
    // (can be updated per round of search)
    // end: index of tail in the current domain!
    // (can be updated)

    x: is input array (fixed), it is also the initial search domain

    key: search key (fixed)
```
Example

```c
int BinarySearch(int* x, int key, int head, int end){

    head = 0

    end = 6

    x = { 1, 3, 4, 7, 9, 10, 12 }.

    key = 9

    Round 1. Search domain is {1, 3, 4, 7, 9, 10, 12}.
```
int BinarySearch(int* x, int key, int head, int end) {

    head = 4  
    end = 6  

Round 1. Search domain is \{1, 3, 4, 7, 9, 10, 12\}.
Round 2. Search domain is \{9, 10, 12\}.
Example

```c
int BinarySearch(int* x, int key, int head, int end){

    If head = end, we have only one element in the domain.

    head = 4  end = 4

    Round 1. Search domain is {1, 3, 4, 7, 9, 10, 12}.

    Round 2. Search domain is {9, 10, 12}.

    Round 3. Search domain is {9}.

    Note x is unchanged.
```

x = {1, 3, 4, 7, 9, 10, 12}.

key = 9
Calculate midpoint by \((\text{head} + \text{end}) / 2\).

```c
int BinarySearch(int* x, int key, int head, int end) {

    head = 0
    end = 6

    \text{midpoint index} = (\text{head} + \text{end}) / 2
    = (0 + 6) / 2
    = 3.
}
Example

```c
int BinarySearch(int* x, int key, int head, int end) {
    head = 4
    end = 6

    midpoint index = (head + end) / 2
    = (4 + 6) / 2
    = 5.
```
Example

```c
int BinarySearch(int* x, int key, int head, int end) {
    head = 4
    end = 4
    midpoint index = (head + end) / 2
    = (4 + 4) / 2
    = 4.
```
Q: what if \((\text{head} + \text{end}) / 2\) is not integer?

```c
int BinarySearch(int* x, int key, int head, int end) {
    head = 1
    end = 6
    midpoint index = (head + end) / 2
    = (1 + 6) / 2
    = 3.5
    Q: pick 7 or 9?
```
Characterize subdomain in binary search using midpoint.

We can use midpoint plus head and end to characterize subdomains in binary search.
Example

midpoint

1  3  4  7  9  10  12

head  midpoint - 1  midpoint + 1  end

left subdomain  right subdomain
Now, let’s look at the body of the recursive function.

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) { return -1; }  // Condition to stop.
    int midpoint = (head + end) / 2;  // Get midpoint.
    if (key == x[midpoint]) { return midpoint; }  // if it is key, return midpoint index
    else if (key < x[midpoint]) {  // if key < mid, pick left subdomain.
        BinarySearch(x, key, head, midpoint - 1);
    } else if (key > x[midpoint]) {  // if key > mid, pick right subdomain.
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```
Q: what if elements are sorted in a descending order?

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}
    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

Condition to stop.

Get midpoint.

if it is key, return midpoint index

if key < mid, pick left subdomain.

Assume ascending order.

if key > mid, pick right subdomain.
Call the recursive binary search function in main.

```cpp
int main()
{
    int x[9] = { 1, 3, 5, 7, 9, 11, 13, 15, 19 };
    cout << BinarySearch(x, 5, 0, 8);
    return 0;
}
```

Output index of the key or -1.

original array, key, head index, tail index
Let’s see a scenario where recursive binary search succeeds!
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\} 
Key = 5

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) { return -1; }

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);  
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

head = 0, end = 8. 

condition (head > end) is false. 

This means domain is not empty.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\} 

Key = 5

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;} // condition (head > end) is false.

    int midpoint = (head + end) / 2; // midpoint = 4, so x[4] = 9.

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}  \quad \text{Key = 5}

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

Head = 0, end = 3.

Condition (head > end) is false.


5 < 9, pick left subdomain.

Call BinarySearch function again, with head = 0 and end = 4-1 = 3.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}  

Key = 5

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) { return -1; }

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    } else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

head = 0, end = 3.  

condition (head > end) is false.  

This means domain is not empty.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}  

Key = 5

\begin{verbatim}
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}

    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
\end{verbatim}
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}                         Key = 5

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}
    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);      
    }
}
```

 condition (head > end) is false.

midpoint = 1, so x[1] = 3.

5 > 3, pick right subdomain.

Call BinarySearch function again, with head = 1+1 = 2 and end = 3.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}  \hspace{1cm} \text{Key} = 5

int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}                 Key = 5

```c
int BinarySearch(int* x, int key, int head, int end){
    if (head > end) { return -1; }  // condition (head > end) is false.
    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

midpoint = 2, so x[2] = 5. Actually, midpoint = 2.5. I choose to pick the smaller one nearest to the midpoint.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\} 

```c
int BinarySearch(int* x, int key, int head, int end){
    if (head > end) {return -1;}  // condition (head > end) is false.

    int midpoint = (head + end) / 2;  // midpoint = 2, so x[2] = 5.

    if (key == x[midpoint]) { return midpoint;  // 5 == 5, key found!
        // return midpoint 2.
    } else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    } else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

Search is done, back to main.
Call the recursive binary search function in main.

```cpp
int main()
{
    int x[9] = { 1, 3, 5, 7, 9, 11, 13, 15, 19 };
    cout << BinarySearch(x, 5, 0, 8);
    return 0;
}
```

Output 2!

original array, key, head index, tail index
Let’s see another scenario where recursive binary search fails!

This is determined by comparing “head” and “end”.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}

Suppose key = 4!

```c
int BinarySearch(int* x, int key, int head, int end){
    if (head > end) {return -1;}

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}

Suppose key = 4!

```cpp
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) { return -1; }

    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}

Suppose key = 4!

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}
    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

head = 2, end = 2.

condition (head > end) is false.


4 < 5, pick left subdomain.

Call BinarySearch function again, with head = 2 and end = 3-1 = 2.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}

Suppose key = 4!

```c
int BinarySearch(int* x, int key, int head, int end){
    if (head > end) {return -1;}

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

head = 2, end = 2.

condition (head > end) is false.

This means domain is not empty.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}

Suppose key = 4!

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) { return -1; }

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    } else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

head = 2, end = 2.

Condition (head > end) is false.


(2 + 2) / 2 = 2. So far so good!
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\}

Suppose key = 4!

```cpp
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) { return -1; }
    int midpoint = (head + end) / 2;
    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {
        BinarySearch(x, key, head, midpoint - 1);
    } else if (key > x[midpoint]) {
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

head = 2, end = 1.

condition (head > end) is false.


4 < 5, pick left subdomain.

Call BinarySearch function again, with head = 2 and end = 2 - 1 = 1.

The domain is now empty, but we still call the function and terminate in the next round.
Initial Domain = \{1, 3, 5, 7, 9, 11, 13, 15, 19\} 

```c
int BinarySearch(int* x, int key, int head, int end) {
    if (head > end) {return -1;}

    int midpoint = (head + end) / 2;

    if (key == x[midpoint]) { return midpoint; }
    else if (key < x[midpoint]) {  
        BinarySearch(x, key, head, midpoint - 1);
    }
    else if (key > x[midpoint]) {  
        BinarySearch(x, key, midpoint + 1, end);
    }
}
```

Suppose key = 4!

head = 2, end = 1. 

condition (head > end) is true.

This means domain is empty!  

return -1 and terminate search.
Call the recursive binary search function in main.

```cpp
int main()
{
    int x[9] = { 1, 3, 5, 7, 9, 11, 13, 15, 19 };

    cout << BinarySearch(x, 4, 0, 8);

    return 0;
}
```

Output -1.
Outline

What is search?

Linear Search Algorithm: examples of success and failure.

Binary Search Algorithm: examples of success and failure, other issues.

Recursive Algorithm: binary search (pseudo-code and code), Fibonacci sequence

Time Complexity and Big O
Generate Fibonacci sequence using a recursive function.

**Fibonacci sequence**: every number is sum of the previous two numbers:

\[
\begin{align*}
2 + 3 & = 5 \\
13 + 21 & = 34 \\
0, & \quad 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad 21, \quad 34, \quad \ldots \quad
\end{align*}
\]
Recursive nature of Fibonacci sequence.

We can generate Fibonacci sequence using recursive function!

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ....

\[ f(...) \]

\[ f \text{ adds two inputs and returns sum.} \]
We can generate Fibonacci sequence using recursive function!

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ....
```

Inside the previous f(), call itself to add the last input and sum.
Example

We can generate Fibonacci sequence using recursive function!

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ……

Inside the previous f(), call itself to add the last input and sum.
Example

We can generate Fibonacci sequence using recursive function!

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ……

Inside the previous f(), call itself to add the last input and sum.
Example recursive function that generates Fibonacci sequence.

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' '; // Output sum
        GenFib(b, sum, maxNum);
    }
}
```

Add two inputs to sum.

If sum is too big, stop.

We do need to set up conditions to terminate a recursive algorithm.

Recursion.

Output sum and call GenFib.

New inputs are b and sum.
Call GenFib in main.

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}

int main() {
    GenFib(0, 1, 1000);
}
```

Function Call

GenFib(0,1,1000).

Start with the first two numbers 0 and 1.

Set maxNum = 1000.
Round 1 of Recursion

```c++
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

<table>
<thead>
<tr>
<th>GenFib(0, 1, 1000);</th>
</tr>
</thead>
</table>

```c++
int main() {
    GenFib(0, 1, 1000);
    sum = 0 + 1 = 1.
}
```

GenFib(0, 1, 1000).
Round 1 of Recursion

```c++
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

```c++
int main()
{
    GenFib(0, 1, 1000);
}
```

GenFib(0, 1, 1000).

sum > maxNum is false, run “else” statement.
Round 1 of Recursion

```c++
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

```c++
int main() {
    GenFib(0, 1, 1000);
}
```

GenFib(0,1,1000).

Output sum = 1.
Call GebFib with inputs $b = 1$, sum = 1.

```
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

```
int main() {
    GenFib(0, 1, 1000);
}
```

Call GebFib with inputs $b = 0$, sum = 1.

```
GenFib(0, 1, 1000).
```

Call GebFib with inputs $b = 1$, sum = 1.

```
GenFib(1, 1, 1000).
```
Round 2 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

```c
int main() {
    GenFib(0, 1, 1000);
    sum = 1 + 1 = 2
    GenFib(1, 1, 1000).
}
Round 2 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

GenFib(0,1,1000).
GenFib(1,1,1000).

sum > maxNum is false, run “else” statement.
Round 2 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```c
int main() {
    GenFib(0, 1, 1000);
}
```

Function Call
GenFib(0,1,1000).
GenFib(1,1,1000).

Output sum = 2.
Round 2 of Recursion

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```cpp
int main() {
    GenFib(0, 1, 1000);
}
```

Function Call

- GenFib(0, 1, 1000).
- GenFib(1, 1, 1000).
- GenFib(1, 2, 1000).

Call GebFib with inputs b = 1, sum = 2.
Round 3 of Recursion

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    }
    else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```cpp
int main()
{
    GenFib(0, 1, 1000);
    GenFib(1, 1, 1000);
    GenFib(1, 2, 1000);
}
```

Function Call
- GenFib(0,1,1000).
- GenFib(1,1,1000).
- GenFib(1,2,1000).

sum = 1 + 2 = 3.
Round 3 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

Function Call

- `GenFib(0, 1, 1000)`
- `GenFib(1, 1, 1000)`
- `GenFib(1, 2, 1000)`
Round 3 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```c
int main() {
    GenFib(0, 1, 1000);
}
```

```
Output sum = 3.
```

Function Call
- GenFib(0,1,1000).
- GenFib(1,1,1000).
- GenFib(1,2,1000).
Round 3 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```c
int main() {
    GenFib(0, 1, 1000);
    GenFib(1, 2, 1000);
    GenFib(2, 3, 1000);
    GenFib(1, 1, 1000);
    GenFib(1, 2, 1000);
    GenFib(2, 3, 1000);
}
```

Call GebFib with inputs

- b = 2, sum = 3.
Continue the recursion until the threshold is hit.
Round 16 of Recursion

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    }
    else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```c
int main()
{
    GenFib(0, 1, 1000);
    sum = 610 + 987 = 1597.
    GenFib(1, 1, 1000).
    GenFib(1, 2, 1000).
    ...
    GenFib(610, 987, 1k).
```
Round 16 of Recursion

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```
int main()
{
    GenFib(0, 1, 1000);
    GenFib(1, 1, 1000);
    GenFib(1, 2, 1000);
    ....
    GenFib(610, 987, 1k);
}
```

sum > maxNum is true!, run "if" statement.
Round 16 of Recursion

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```
int main()
{
    GenFib(0, 1, 1000);
    GenFib(1, 1, 1000);
    GenFib(1, 2, 1000);
    ...
    GenFib(610, 987, 1k).
```

"return" will terminate the recursion.
End recursion and go back to main.

```c
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```c
int main() {
    GenFib(0, 1, 1000);
    GenFib(1, 1, 1000);
    GenFib(1, 2, 1000);
    ....
    GenFib(610, 987, 1k);
}
```

Get back to the main program and run the rest.
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}

int main()
{
    GenFib(0, 1, 1000);
}

1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
Another example output with smaller cutoff (100).

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}

int main() {
    GenFib(0, 1, 100);
    return 0;
}
```

1 2 3 5 8 13 21 34 55 89
Another example output with larger initial inputs (5, 8).

```cpp
int GenFib(int a, int b, int maxNum) {
    int sum = a + b;
    if (sum > maxNum) {
        return 0;
    } else {
        cout << sum << ' ';
        GenFib(b, sum, maxNum);
    }
}
```

```cpp
int main()
{
    GenFib(5, 8, 1000);
    return 0;
}
```

```
13 21 34 55 89 144 233 377 610 987
```
Outline

What is search?

Linear Search Algorithm: examples of success and failure.

Binary Search Algorithm: examples of success and failure, other issues.

Recursive Algorithm: binary search (pseudo-code and code), Fibonacci sequence

Time Complexity and Big O