CSCI 241

Scott Wehrwein

Runtime Analysis:
Case study - Binary Search
Best-, Worst-, and Average-case Analysis
Goals

Understand the runtime analysis of binary search.

Know how to perform best-case, worst-case, and average-case runtime analysis.
Runtime of Binary Search

Let \( N = A.\text{length} \) and assume \( x \) is not in \( A \).

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    // invariant: \( A[start] \leq x \leq A[end-1] \)
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Let \( N = A.\text{length} \) and assume \( x \) is not in \( A \).

Strategy:
1. Identify constant-time operations.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}

Let N = A.length and assume x is not in A.

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let \( N = A.\text{length} \) and assume \( x \) is not in \( A \).

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let \( N = A.\text{length} \) and assume \( x \) is not in \( A \).

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    // invariant: \( A[start] \leq x \leq A[end-1] \)
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let $N = A\.\text{length}$ and assume $x$ is not in $A$.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A\.length;
    // invariant: $A[\text{start}] \leq x \leq A[\text{end}-1]$
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:

1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.

L* OR [-0x253](L+1)*
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.

Total: $5L + 4$
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Total: $5L + 4$

so, O(L) ...but what is L?
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:

\[
\begin{array}{c|c}
A & \text{[Grey]} \\
\end{array}
\]
• Steps of a hypothetical binary search:

A can be here can't be here
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
Steps of a hypothetical binary search:
• Steps of a hypothetical binary search:
Size of the "can be here" region:

A can be here

A can be here

A cbh

A cbh

A cbh

A
Size of the "can be here" region:

<table>
<thead>
<tr>
<th>A</th>
<th>can be here</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>can be here</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>cbh</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>cbh</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>cbh</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Size of the "can be here" region:

\[
\begin{array}{c|c|c}
A & \text{can be here} & N \\
A & \text{can be here} & N/2 \\
A & \text{cbh} & \\
A & \text{cbh} & \\
A & \text{cbh} & \\
A & \text{can be here} & \\
\end{array}
\]
Size of the "can be here" region:

\[ \frac{N}{2} \]
Size of the "can be here" region:

A can be here

A can be here

A cbh

A cbh

A cbh

A
Size of the "can be here" region:

\[ \frac{N}{8} \]

\[ \frac{N}{4} \]

\[ \frac{N}{2} \]

\[ N \]

integer division
Size of the "can be here" region:

\[ \frac{N}{2} \]

\[ \frac{N}{4} \]

\[ \frac{N}{8} \]

\[ \frac{N}{16} \]
Size of the "can be here" region:

A  can be here

A  can be here

A  cbh

A  cbh

A  cbh

A  cbh

A  cbh

N  \frac{N}{2}

N  \frac{N}{4}

N  \frac{N}{8}

N  \frac{N}{16}

N  \frac{N}{32} = 0

integer division
Size of the "can be here" region:

\[
\begin{align*}
A & \quad \text{can be here} \\
A & \quad \text{can be here} \\
A & \quad \text{cbh} \\
A & \quad \text{cbh} \\
A & \quad \text{cbh} \\
A & \quad \text{cbh} \\
\end{align*}
\]

\[L = 5\]

\[\frac{N}{N/2} = \frac{N}{2}\]
\[\frac{N}{N/4} = \frac{N}{4}\]
\[\frac{N}{N/8} = \frac{N}{8}\]
\[\frac{N}{N/16} = \frac{N}{16}\]
\[\frac{N}{N/32} = 0\]

integer division
So, \( L \) is the answer to:
How many times can I divide \( N \) by 2 before it becomes 0?
How many times can I divide $N$ by 2 before it becomes 0?

Or equivalently, using real (non-integer) division:

How many times can I divide $N$ by 2 before it's less than 1?

\[
\frac{N}{2^k} < 1
\]

\[
N < 2^k
\]

\[
\log_2 N < \log_2 2^k
\]

\[
\log_2 N < k
\]

\[
\left\lfloor \log_2 N \right\rfloor = k
\]
Runtime of Binary Search

Let \( N = A\.length \) and assume \( x \) is not in \( A \).

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    // Total: 5L + 4
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
3. Drop constants and lower-order terms.
Runtime of Binary Search

Let N = A.length and assume x is not in A.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Strategy:
1. Identify constant-time operations.
2. Determine how many times each happens.
3. Drop constants and lower-order terms.

Total: $5\log_2(n) + 4$
Runtime of Binary Search

Let $N = A.length$ and assume $x$ is not in $A$.

```java
public static int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    // invariant: $A[start] \leq x \leq A[end-1]$
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

Total: $5L + 4$

$5 \log_2(n) + 4$ is $O(\log n)$
Aside: Bases of Logarithms

Fact: converting from one base to another only requires multiplication by a constant.

$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$$

$$\log_b(x) = \frac{1}{\log_d(b)} \cdot \log_d(x)$$
Aside: Bases of Logarithms

Corollary: the base of the logarithm doesn't affect the big-O class.

\[ C \cdot \log_2(n) = \log_{10} n \]

\( O(\log_2 n) \) is same as \( O(\log_{10} n) \)
Aside: Bases of Logarithms

Convention: We can use logs without specifying a base in big-O notation.

\[ \Theta(\log_2 n) \]

\[ \Theta(\log n) \]
Which algorithm is better?

Suppose you have two different algorithms that solve the same problem. For example, search a sorted array.

```java
int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++){
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

```java
int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else {
            if (x < A[mid]) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }
    }
    return -1;
}
```

A consequential question:
Which is better?
What is "better"?
Which algorithm is better?

Suppose you have two different algorithms that solve the same problem. For example, search a sorted array.

```java
int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

```java
int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

A consequential question:
Which is better?
What is "better"?
O(n)
Which algorithm is better?

Suppose you have two different algorithms that solve the same problem. For example, search a sorted array.

```java
int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}

int binarySearch(int[] A, int x) {
    int start = 0;
    int end = A.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            end = mid;
        } else {
            start = mid + 1;
        }
    }
    return -1;
}
```

A consequential question:
Which is better?
What is "better"?

O(n)          O(log n)
The operation count depends on the data!

- If \( x \) is at \( A[0] \), runtime is \( O(1) \)
- If \( x \) is not in \( A \), runtime is \( O(N) \)
- If \( x \) is at a random location in \( A \), runtime is \( O(N) \)
Best-, worst-, and Average-case

/** Return the index of x in A or -1 not found. */
public static int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}

The operation count depends on the data!
• If x is at A[0], runtime is O(1) (best-case runtime)
• If x is not in A, runtime is O(N)
• If x is at a random location in A, runtime is O(N)
** Best-, worst-, and Average-case **

```java
/** Return the index of x in A or -1 not found. */
public static int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

The operation count depends on the data!

- If \( x \) is at \( A[0] \), runtime is \( O(1) \) (best-case runtime)
- If \( x \) is not in \( A \), runtime is \( O(N) \) (worst-case runtime)
- If \( x \) is at a random location in \( A \), runtime is \( O(N) \)
Best-, worst-, and Average-case

```java
/** Return the index of x in A or -1 not found. */
public static int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

The operation count depends on the data!

- If `x` is at `A[0]`, runtime is O(1) (best-case runtime)
- If `x` is not in `A`, runtime is O(N) (worst-case runtime)
- If `x` is at a random location in `A`, runtime is O(N)  
  (one possible notion of average-case runtime)