Summary previous lecture: binary search

Linear search

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

Binary search

\[ s = \log_2(N) \]

<table>
<thead>
<tr>
<th>N</th>
<th>Linear search (s=N/2)</th>
<th>(\log_2(N))</th>
<th>Binary search (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>3.32</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>6.64</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>500</td>
<td>9.97</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>5,000</td>
<td>13.3</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>50,000</td>
<td>16.6</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>500,000</td>
<td>19.9</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>5,000,000</td>
<td>23.3</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>50,000,000</td>
<td>26.6</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>500,000,000</td>
<td>29.9</td>
<td>30</td>
</tr>
</tbody>
</table>

Logarithm growth rate \(<\) Linear growth rate
Summary previous lecture: Big O Notation

- Indicate how the running time $T$ is affected by the number of items $N$.
- The Big O notation introduces the letter $O$ which means “order of”
  - Cost $O(1)<O(\log N)<O(N)$

**TABLE 2.5 Running Times in Big O Notation**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running Time in Big O Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary search</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Insertion in unordered array</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion in ordered array</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Deletion in unordered array</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Deletion in ordered array</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
Big O Notation and Running Time

- **Algorithm A is O(N)** ...For a given $N$, Algo A runs in $x$ seconds
  - If $N \leftarrow 2*N$, it would run in $2*x$ seconds
  - If $N \leftarrow 10*N$, it would run in $10*x$ seconds
  - If $N \leftarrow N*N$, it would run in $N*x$ seconds

- **Algorithm B is O(logN)**.. For a given $N$ (large), Algo B runs in $x$ seconds
  - If $N \leftarrow 2*N$, it would run in $\sim x$ seconds
  - If $N \leftarrow 10*N$, it would run in $\sim x$ seconds
  - If $N \leftarrow N*N$, it would run in $2*x$ seconds

- **Algorithm C is O(N²)**... For a given $N$, Algo C runs in $x$ seconds
  - If $N \leftarrow 2*N$, it would run in $4*x$ seconds
  - If $N \leftarrow 10*N$, it would run in $100*x$ seconds
  - If $N \leftarrow N*N$, it would run in $N^2*x$ seconds

**Cost:** Algo B < Algo A < Algo C

$O(\log N) < O(N) < O(N^2)$
Big O Notation

The graph illustrates the relationship between the number of steps and the number of items (N), showing different complexities:

- **O(1)**: Constant time complexity, no matter how many items there are.
- **O(log N)**: Logarithmic time complexity, grows slowly as the number of items increases.
- **O(N)**: Linear time complexity, grows proportionally to the number of items.
- **O(N^2)**: Quadratic time complexity, grows very quickly as the number of items increases.

The graph also indicates how to improve running time by keeping the complexity low.
Big O Notation – Asymptotic Analysis

- Constants and lower degrees are ignored
- Capture the complexity for large $N$
  - $N/2 \quad O(N)$
  - $3N^2+15N \quad O(N^2)$
  - $0.00001*N^2+15000*N \quad O(N^2)$
  - $N^2*N+10*N^2\log(N) \quad O(N^3)$
  - $123+\log(321) \quad O(1)$
  - $(N+\log(N))^2 \quad O(N^2)$
  - $N*(5+\log(N)) \quad O(N\log N)$
  - $1+2+3+\ldots+N = N(N+1)/2 \quad O(N^2)$
Example 1:

\[
\{i=0,i=1,i=2,i=3,...,i=N\} \quad \Rightarrow \quad O(N)
\]

Example 2:

\[
N \text{ steps for outer loop, } N \text{ steps for inner loop} \quad \Rightarrow \quad O(N^2)
\]

Example 3:

\[
\{i=N,i=N/2,i=N/4,i=N/8,...,i=1\} \quad \Rightarrow \quad \log_2(N) \text{ steps is } O(\log N)
\]

\[
\{i=1,i=2,i=4,i=8,i=16,..,i=N\} \quad \Rightarrow \quad \log_2(N) \text{ steps is } O(\log N)
\]
Big O Notation - Examples

- **Example 5**

  ```java
  for (int i=0; i<N; i++){
    for (int j=0; j<10000; j++){
      //do something
    }
  }
  ```

  - N steps outer loop, inner loop constant number of steps  \( \equiv O(N) \)

- **Example 4**

  ```java
  for (int i=0; i<N; i++){
    for (int j=0; j<i; j++){ //do something
  }
  ```

  - Number of steps \( s \) in inner loop

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>N-1</td>
</tr>
</tbody>
</table>

  - Summing all the inner loop steps

  \[
  0 + 1 + 2 + 3 + \ldots + (N-1) = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} \equiv O(N^2)
  \]
Practice (from Fall15 midterm)….

```java
for (int i=0; i<N; i=i+2) {
    //do something
}
```

```java
for (int i=0; i<100; i++) {
    //do something
}
```

```java
for (int i=0; i<N; i++) {
    for (int j=0; j<i; j++) {
        for (int k=0; k<j; k++) {
            //do something
        }
    }
}
```

```java
for (int i=1; i<N; i=i*5) {
    //do something
}
```

```java
for (int i=0; i<N; i++) {
    for (int j=N; j>0; j=j/2) {
        //do something
    }
}
```

//here we consider two variables input N and M
for (int i=1; i<N; i=i*2) {
    for (int j=0; j<M; j++) {
        //do something
    }
}
### Classes of Algorithm Complexity

- **O(1)** constant
- **O(logN)** logarithmic growth
- **O(N)** linear growth

- **O(NlogN)** loglinear growth
- **O(N^2)** quadratic growth
- **O(N^3)** cubic growth
- **O(2^N)** exponential growth

![Graph showing the growth of different classes of algorithm complexity](image)

- **Cost**
- **O(2^N)**
- **O(N^3)**
- **O(N^2)**
- **O(NlogN)**
- **O(N)**

- **Problem size N**
- **O(logN)**
- **O(I)**
Cost analysis: $O(1)<O(\log N)<O(N)<O(N^2)<O(N^3)<O(2^n)$

More to the story (concepts beyond this class but good to have in mind)

- N small: $O(N^3)$ algorithms can better exploit data locality in memory
- N large: $O(N)$ algorithms most likely respond better to parallelism
Arrays: Take Home

- **Array have fixed size** – good estimate of the number of items needed
- A higher level class can be designed to make things simple for the class user (handles privately indexes and number of items)
- Unordered arrays offer
  - fast insertion $O(1)$
  - but slow linear search and deletion $O(N)$
- Ordered arrays arrays offer
  - fast binary search $O(\log N)$
  - but slow insertion and deletion $O(N)$
- **Ordered array** useful in situation where once the array is ordered, search are very frequent but insertion/deletion are not
  - Example: search words in dictionary
- The **Big O notation** provides information about the algorithm complexity, relative running time performances between algorithms