Design & Analysis of Algorithms
The Big O
Lecture 18
How it scales
How it scales

In analyzing running time (or memory/power consumption) of an algorithm, we are interested in how it scales as the problem instance grows in “size”
How it scales

In analyzing running time (or memory/power consumption) of an algorithm, we are interested in how it scales as the problem instance grows in “size”

Running time on small instances of a problem are anyway small. Not a serious concern.
How it scales

- In analyzing running time (or memory/power consumption) of an algorithm, we are interested in how it scales as the problem instance grows in “size”

- Running time on small instances of a problem are anyway small. Not a serious concern.

- Also, exact time/number of steps is less interesting
How it scales

In analyzing running time (or memory/power consumption) of an algorithm, we are interested in how it scales as the problem instance grows in “size”

Running time on small instances of a problem are anyway small. Not a serious concern.

Also, exact time/number of steps is less interesting

Can differ in different platforms (by “constant factors” independent of input size). Not a property of the algorithm alone.
How it scales

In analyzing running time (or memory/power consumption) of an algorithm, we are interested in how it scales as the problem instance grows in “size”

Running time on small instances of a problem are anyway small. Not a serious concern.

Also, exact time/number of steps is less interesting

Can differ in different platforms (by “constant factors” independent of input size). Not a property of the algorithm alone.

Thus “unit” of time typically ignored
How it scales
How it scales

Interested in how a function scales with its input: behavior on large values, up to constant factors
How it scales

Interested in how a function scales with its input: behavior on large values, up to constant factors

e.g., suppose number of “steps” taken by an algorithm to sort a list of n elements varies between 3n and 4n²+9 (depending on what the list looks like)
How it scales

Interested in how a function **scales** with its input: behavior on **large values**, up to **constant factors**

e.g., suppose number of “steps” taken by an algorithm to sort a list of n elements varies between 3n and 4n²+9 (depending on what the list looks like)

If n is doubled, time taken could become (roughly, in the worst case) 4 times. If n is tripled, it could become (roughly, in the worst case) 9 times
How it scales

Interested in how a function scales with its input: behavior on large values, up to constant factors.

e.g., suppose number of “steps” taken by an algorithm to sort a list of n elements varies between $3n$ and $4n^2+9$ (depending on what the list looks like).

If $n$ is doubled, time taken could become (roughly, in the worst case) 4 times. If $n$ is tripled, it could become (roughly, in the worst case) 9 times.

An upperbound that grows “like” $n^2$.
Upperbounds: Big O
Upperbounds: Big O

$T(n)$ has an upperbound that grows "like" $f(n)$
Upperbounds: Big O

- $T(n)$ has an upperbound that grows "like" $f(n)$
- $T(n) = O(f(n))$
Upperbounds: Big O

- \( T(n) \) has an upperbound that grows “like” \( f(n) \)
- \( T(n) = O(f(n)) \)

Unfortunate notation! Sometimes, \( T(n) \in O(f(n)) \)
Upperbounds: Big O

- $T(n)$ has an upperbound that grows "like" $f(n)$

  $T(n) = O(f(n))$

- $\exists c > 0, k \in \mathbb{Z}^+, \forall n \geq k, \, 0 \leq T(n) \leq c \cdot f(n)$

Unfortunate notation!
Sometimes, $T(n) \in O(f(n))$
Upperbounds: Big O

- $T(n)$ has an upperbound that grows “like” $f(n)$
  - $T(n) = O(f(n))$
  - $\exists c > 0, k \in \mathbb{Z}^+, \forall n \geq k, \ 0 \leq T(n) \leq c \cdot f(n)$

- Note: we are defining it only for $T$ & $f$ which eventually stay non-negative

Unfortunate notation!
Sometimes, $T(n) \in O(f(n))$
Upperbounds: Big O

- T(n) has an upperbound that grows “like” f(n)
  - $T(n) = O(f(n))$
  - $\exists c > 0, k \in \mathbb{Z}^+, \forall n \geq k, 0 \leq T(n) \leq c \cdot f(n)$

Note: we are defining it only for T & f which eventually stay non-negative

Note: order of quantifiers! c can’t depend on n

Unfortunate notation! Sometimes, $T(n) \in O(f(n))$
Upperbounds: Big $O$

- $T(n)$ has an upperbound that grows "like" $f(n)$
  - $T(n) = O(f(n))$

- $\exists c > 0, k \in \mathbb{Z}^+, \forall n \geq k, \ 0 \leq T(n) \leq c \cdot f(n)$

- Note: we are defining it only for $T$ & $f$ which eventually stay non-negative

- Note: order of quantifiers! $c$ can't depend on $n$

- Important: If $T(n) = O(f(n))$, $f(n)$ could be much larger than $T(n)$ (or roughly, up to a constant, smaller than $T(n)$)
Big O examples
Big O examples

Suppose $T(n) = 7n^2$
Big O examples

Suppose $T(n) = 7n^2$

- e.g., $T(n) = O(n^3)$. $\forall n \geq 7$, $T(n) \leq 1 \cdot n^3$
Suppose $T(n) = 7n^2$

e.g., $T(n) = O(n^3)$. $\forall n \geq 7, T(n) \leq 1 \cdot n^3$

Also, $T(n) = O(n^2)$. $\forall n \geq 1, T(n) \leq 7 \cdot n^2$
Big O examples

Suppose \( T(n) = 7n^2 \)

- e.g., \( T(n) = O(n^3) \). \( \forall n \geq 7, T(n) \leq 1 \cdot n^3 \)
- Also, \( T(n) = O(n^2) \). \( \forall n \geq 1, T(n) \leq 7 \cdot n^2 \)
- But \( T(n) \neq O(n) \). \( \forall c > 0, \forall k > 0, \exists n^* \geq k \ T(n^*) > c \cdot n^* \)
Big O examples

Suppose $T(n) = 7n^2$

- e.g., $T(n) = O(n^3)$. $\forall n \geq 7$, $T(n) \leq 1 \cdot n^3$
- Also, $T(n) = O(n^2)$. $\forall n \geq 1$, $T(n) \leq 7 \cdot n^2$
- But $T(n) \neq O(n)$. $\forall c > 0$, $\forall k > 0$, $\exists n^* \geq k$ $T(n^*) > c \cdot n^*$

e.g., $n^* = \max(k, c)$
**Big O examples**

- Suppose $T(n) = 7n^2$
  - e.g., $T(n) = O(n^3)$. $\forall n \geq 7$, $T(n) \leq 1 \cdot n^3$
  - Also, $T(n) = O(n^2)$. $\forall n \geq 1$, $T(n) \leq 7 \cdot n^2$
  - But $T(n) \neq O(n)$. $\forall c > 0$, $\forall k > 0$, $\exists n^* \geq k$ $T(n^*) > c \cdot n^*$

- Suppose $T(n) = 14n + 2$
  - e.g., $n^* = \max(k, c)$
Big O examples

Suppose $T(n) = 7n^2$

- e.g., $T(n) = O(n^3)$. $\forall n \geq 7, T(n) \leq 1 \cdot n^3$
- Also, $T(n) = O(n^2)$. $\forall n \geq 1, T(n) \leq 7 \cdot n^2$
- But $T(n) \neq O(n)$. $\forall c > 0, \forall k > 0, \exists n^* \geq k \quad T(n^*) > c \cdot n^*$

Suppose $T(n) = 14n + 2$

- $T(n) = O(n)$. Also $T(n) = O(n^2)$.

E.g., $n^* = \max(k, c)$
Big O examples
Big O examples

Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$
Big O examples

Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$

i.e., $\forall n \geq k_T, \ 0 \leq T(n) \leq c_T \cdot f(n)$ and $\forall n \geq k_R, \ 0 \leq R(n) \leq c_R \cdot f(n)$
Big O examples

Suppose \( T(n) = O(f(n)) \) and \( R(n) = O(f(n)) \)

i.e., \( \forall n \geq k_T, \ 0 \leq T(n) \leq c_T \cdot f(n) \) and \( \forall n \geq k_R, \ 0 \leq R(n) \leq c_R \cdot f(n) \)

\( T(n) + R(n) = O(f(n)) \)
Big O examples

Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$

i.e., $\forall n \geq k_T, 0 \leq T(n) \leq c_T \cdot f(n)$ and $\forall n \geq k_R, 0 \leq R(n) \leq c_R \cdot f(n)$

$T(n) + R(n) = O(f(n))$

Then, $\forall n \geq \max(k_T, k_R), 0 \leq T(n) + R(n) \leq (c_R + c_T) \cdot f(n)$
Big O examples

Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$

i.e., $\forall n \geq k_T, 0 \leq T(n) \leq c_T \cdot f(n)$ and $\forall n \geq k_R, 0 \leq R(n) \leq c_R \cdot f(n)$

$T(n) + R(n) = O(f(n))$

Then, $\forall n \geq \max(k_T,k_R), 0 \leq T(n)+R(n) \leq (c_R+c_T) \cdot f(n)$

If eventually ($\forall n \geq k$), $T(n) \geq R(n)$, then $T(n) - R(n) = O(T(n))$
Big O examples

Suppose \( T(n) = O(f(n)) \) and \( R(n) = O(f(n)) \)

i.e., \( \forall n \geq k_T, 0 \leq T(n) \leq c_T \cdot f(n) \) and \( \forall n \geq k_R, 0 \leq R(n) \leq c_R \cdot f(n) \)

\( T(n) + R(n) = O(f(n)) \)

Then, \( \forall n \geq \max(k_T,k_R), 0 \leq T(n)+R(n) \leq (c_R+c_T) \cdot f(n) \)

If eventually (\( \forall n \geq k \)), \( T(n) \geq R(n) \), then \( T(n) - R(n) = O(T(n)) \)

\( \forall n \geq \max(k,k_R), T(n)-R(n) \leq 1 \cdot T(n) \)
Big O examples

Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$

i.e., $\forall n \geq k_T, 0 \leq T(n) \leq c_T \cdot f(n)$ and $\forall n \geq k_R, 0 \leq R(n) \leq c_R \cdot f(n)$

$T(n) + R(n) = O(f(n))$

Then, $\forall n \geq \max(k_T, k_R), 0 \leq T(n) + R(n) \leq (c_R + c_T) \cdot f(n)$

If eventually ($\forall n \geq k$), $T(n) \geq R(n)$, then $T(n) - R(n) = O(T(n))$

$\forall n \geq \max(k, k_R), T(n) - R(n) \leq 1 \cdot T(n)$

e.g., $7n^2 + 14n + 2 = O(n^2)$ because $7n^2, 14n, 2$ are all $O(n^2)$
Big O examples

- Suppose $T(n) = O(f(n))$ and $R(n) = O(f(n))$

  - i.e., $\forall n \geq k_T$, $0 \leq T(n) \leq c_T \cdot f(n)$ and $\forall n \geq k_R$, $0 \leq R(n) \leq c_R \cdot f(n)$
  - $T(n) + R(n) = O(f(n))$

  - Then, $\forall n \geq \max(k_T, k_R)$, $0 \leq T(n) + R(n) \leq (c_R + c_T) \cdot f(n)$

- If eventually ($\forall n \geq k$), $T(n) \geq R(n)$, then $T(n) - R(n) = O(T(n))$

  - $\forall n \geq \max(k, k_R)$, $T(n) - R(n) \leq 1 \cdot T(n)$

- e.g., $7n^2 + 14n + 2 = O(n^2)$ because $7n^2$, $14n$, $2$ are all $O(n^2)$

- If $T(n)$ is a degree $d$ polynomial with a positive coefficient for $n^d$, then $T(n) = O(n^d)$
Some important functions
Some important functions

$T(n) = O(1): \exists c \text{ s.t. } T(n) \leq c \text{ for all sufficiently large } n$
Some important functions

- $T(n) = O(1)$: $\exists c$ s.t. $T(n) \leq c$ for all sufficiently large $n$
- $T(n) = O(\log n)$. $T(n)$ grows quite slowly, because $\log n$ grows quite slowly (when $n$ doubles, $\log n$ grows by 1)
Some important functions

- $T(n) = O(1)$: $\exists c$ s.t. $T(n) \leq c$ for all sufficiently large $n$
- $T(n) = O(\log n)$. $T(n)$ grows quite slowly, because $\log n$ grows quite slowly (when $n$ doubles, $\log n$ grows by 1)
- $T(n) = O(n)$: $T(n)$ is (at most) linear in $n$
Some important functions

- $T(n) = O(1)$: $\exists c$ s.t. $T(n) \leq c$ for all sufficiently large $n$
- $T(n) = O(\log n)$. $T(n)$ grows quite slowly, because $\log n$ grows quite slowly (when $n$ doubles, $\log n$ grows by 1)
- $T(n) = O(n)$: $T(n)$ is (at most) linear in $n$
- $T(n) = O(n^2)$: $T(n)$ is (at most) quadratic in $n$
Some important functions

* T(n) = O(1): \( \exists c \text{ s.t. } T(n) \leq c \) for all sufficiently large n

* T(n) = O(log n). T(n) grows quite slowly, because log n grows quite slowly (when n doubles, log n grows by 1)

* T(n) = O(n): T(n) is (at most) linear in n

* T(n) = O(n^2): T(n) is (at most) quadratic in n

* T(n) = O(n^d) for some fixed d: T(n) is (at most) polynomial in n
Some important functions

- $T(n) = O(1)$: There exists a constant $c$ such that $T(n) \leq c$ for all sufficiently large $n$.
- $T(n) = O(\log n)$. $T(n)$ grows quite slowly, because $\log n$ grows quite slowly (when $n$ doubles, $\log n$ grows by 1).
- $T(n) = O(n)$: $T(n)$ is (at most) linear in $n$.
- $T(n) = O(n^2)$: $T(n)$ is (at most) quadratic in $n$.
- $T(n) = O(n^d)$ for some fixed $d$: $T(n)$ is (at most) polynomial in $n$.
- $T(n) = O(2^d \cdot n)$ for some fixed $d$: $T(n)$ is (at most) exponential in $n$. 
Question
Below $n$ denotes the number of nodes in a complete and full $m$-ary tree and $h$ its height. Which of the following is/are true?

1. $h = O(\log_m n)$
2. $h = O(\log_2 n)$
3. $n = O(m^h)$
4. $n = O(2^h)$

A. 1 & 3 only
B. 2 & 4 only
C. 1, 3 & 4 only
D. 1, 2 & 3 only
E. 1, 2, 3 & 4
Theta Notation
If we can give a “tight” upper and lower-bound we use the Theta notation.
If we can give a “tight” upper and lower-bound we use the Theta notation

\[ T(n) = \Theta(f(n)) \text{ if } T(n) = O(f(n)) \text{ and } f(n) = O(T(n)) \]
Theta Notation

If we can give a “tight” upper and lower-bound we use the Theta notation

\[ T(n) = \Theta(f(n)) \text{ if } T(n) = O(f(n)) \text{ and } f(n) = O(T(n)) \]

e.g., \( 3n^2 - n = \Theta(n^2) \)
If we can give a “tight” upper and lower-bound we use the Theta notation

\[ T(n) = \Theta(f(n)) \text{ if } T(n) = O(f(n)) \text{ and } f(n) = O(T(n)) \]

E.g., \[ 3n^2 - n = \Theta(n^2) \]

If \[ T(n) = \Theta(f(n)) \text{ and } R(n) = \Theta(f(n)), \] then \[ T(n) + R(n) = \Theta(f(n)) \]
Question
Question

Which of the following is/are true?

1. If \( f(x) = O(g(x)) \) and \( g(x) = O(h(x)) \) then \( f(x) = O(h(x)) \)
2. If \( f(x) = O(g(x)) \) and \( h(x) = O(g(x)) \) then \( f(x) = O(h(x)) \)
3. If \( f(x) = \Theta(g(x)) \) and \( h(x) = \Theta(g(x)) \) then \( f(x) = \Theta(h(x)) \)

A. 1 only
B. 1 & 2 only
C. 3 only
D. 1 & 3 only
E. 1, 2 & 3
Analyzing Algorithms
Analyzing Algorithms

- Analyze correctness and running time (or other resources)
Analyzing Algorithms

- Analyze correctness and running time (or other resources)
- Latter can be quite complicated
Analyzing Algorithms

- Analyze correctness and running time (or other resources)
  - Latter can be quite complicated
- Behavior depends on the particular inputs, but we often restrict the analysis to worst-case over all possible inputs of the same “size”
Analyzing Algorithms

- Analyze correctness and running time (or other resources)
  - Latter can be quite complicated

- Behavior depends on the particular inputs, but we often restrict the analysis to **worst-case** over all possible inputs of the same “size”

- Size of a problem is defined in some natural way (e.g., number of elements in a list to be sorted, number of nodes in a graph to be colored, etc.)
Analyzing Algorithms

- Analyze correctness and running time (or other resources)
  - Latter can be quite complicated

- Behavior depends on the particular inputs, but we often restrict the analysis to worst-case over all possible inputs of the same "size"

- Size of a problem is defined in some natural way (e.g., number of elements in a list to be sorted, number of nodes in a graph to be colored, etc.)

- Generically, could define as number of bits needed to write down the input
Loops
Loops

If the algorithm is “straight-line” without loops or recursion, it would be $O(1)$.
Loops

- If the algorithm is “straight-line” without loops or recursion, it would be $O(1)$
- Need to analyze how many times a loop is taken
Loops

If the algorithm is “straight-line” without loops or recursion, it would be $O(1)$

Need to analyze how many times a loop is taken

e.g. find max among n numbers in an array $L$

```cpp
findmax(L,n) {
    max = L[1]
    for i = 2 to n {
        if (L[i] > max)
            max = L[i]
    }
    return max
}
```
Loops

If the algorithm is “straight-line” without loops or recursion, it would be $O(1)$

Need to analyze how many times a loop is taken

e.g. find max among $n$ numbers in an array $L$

```c
findmax(L,n) {
    max = L[1]
    for i = 2 to n {
        if (L[i] > max)
            max = L[i]
    }
    return max
}
```

Time taken by $\text{findmax}(L,n)$

$T(n) = O(n)$
Nested Loops
Nested Loops

If an outer-loop is executed $p$ times, and each time an inner-loop is executed $q$ times, the code inside the inner-loop is executed $p \cdot q$ times in all.
Nested Loops

- If an outer-loop is executed \( p \) times, and each time an inner-loop is executed \( q \) times, the code inside the inner-loop is executed \( p \cdot q \) times in all.

- The number of times the inner-loop is taken can be different at different executions of the outer-loop.
Nested Loops

If an outer-loop is executed p times, and each time an inner-loop is executed q times, the code inside the inner-loop is executed \( p \cdot q \) times in all.

The number of times the inner-loop is taken can be different at different executions of the outer-loop.

E.g.
```java
for i = 1 to n {
    for j = 1 to i {
        tap-fingers()
    }
}
```
Nested Loops

If an outer-loop is executed $p$ times, and each time an inner-loop is executed $q$ times, the code inside the inner-loop is executed $p \cdot q$ times in all.

The number of times the inner-loop is taken can be different at different executions of the outer-loop.

e.g.

```plaintext
for i = 1 to n {
    for j = 1 to i {
        tap-fingers()
    }
}
```
Nested Loops

If an outer-loop is executed $p$ times, and each time an inner-loop is executed $q$ times, the code inside the inner-loop is executed $p \cdot q$ times in all.

The number of times the inner-loop is taken can be different at different executions of the outer-loop.

E.g.

```plaintext
for i = 1 to n {
    for j = 1 to i {
        tap-fingers()
    }
}
```

What all values of $(i,j)$ are possible when we get here?

$i=1$: $j=1$. $i=2$: $j=1,2$. $i=3$: $j=1,2,3$. ... $i=n$: $j=1,2,\ldots,n$. 
**Nested Loops**

- If an outer-loop is executed \( p \) times, and each time an inner-loop is executed \( q \) times, the code inside the inner-loop is executed \( p \cdot q \) times in all.

- The number of times the inner-loop is taken can be different at different executions of the outer-loop.

- e.g.
  ```java
  for i = 1 to n {
    for j = 1 to i {
      tap-fingers()
    }
  }
  ```

  what all values of \((i,j)\) are possible when we get here?

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
</tr>
<tr>
<td>3</td>
<td>1,2,3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1,2,...,n</td>
</tr>
</tbody>
</table>

  \[ 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = O(n^2) \]
Loops
Loops

\[ i = 1 \]

\[ \text{while } i \leq n \{ \]
  \[ \text{for } j = 1 \text{ to } n \{ \]
    \[ \text{tap-fingers()} \]
  \[ \}
  \]
\[ i = 2 \times i \]

\[ \}\]

\[ \]
Loops

\( i = 1 \)

while \( i \leq n \) {
    for \( j = 1 \) to \( n \) {
        tap-fingers()
    }
    \( i = 2 \times i \)
}

\( i=1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor} \) (\( j=1, 2, \ldots, n \) always)
Loops

\[
\begin{align*}
i & = 1 \\
\text{while } i \leq n \{ \\
& \quad \text{for } j = 1 \text{ to } n \{ \\
& \quad \quad \text{tap-fingers()} \\
& \quad \} \\
& \quad i = 2*i \\
\} 
\end{align*}
\]

\(n\log n\) (j=1,2,\ldots,n always)

\(O(n \log n)\)
Loops

\[ i = 1 \]
while \( i \leq n \) {
    for \( j = 1 \) to \( n \) {
        tap-fingers()
    }
    \( i = 2 \times i \)
}

\[ i = 1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor} \quad (j=1,2,\ldots,n \text{ always}) \]

\[ O(n \log n) \]
Loops

\[ i = 1 \]
\[
\text{while } i \leq n \{
    \text{for } j = 1 \text{ to } n \{
        \text{tap-fingers()}
    \}
    i = 2 \times i
\}
\]

\[ i = 1 \]
\[
\text{while } i \leq n \{
    \text{for } j = 1 \text{ to } i \{
        \text{tap-fingers()}
    \}
    i = 2 \times i
\}
\]

\[ i = 1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor} \quad (j=1,2,\ldots,n \text{ always}) \]
\[ O(n \log n) \]

\[ i = 1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor} \] but i value of j
Loops

\( i = 1 \)

while \( i \leq n \) {
  for \( j = 1 \) to \( n \) {
    tap-fingers()
  }
  \( i = 2 \times i \)
}

\( i = 1 \)

while \( i \leq n \) {
  for \( j = 1 \) to \( i \) {
    tap-fingers()
  }
  \( i = 2 \times i \)
}

\( i = 1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor} \) \( (j=1,2,\ldots,n \text{ always}) \)

\( O(n \log n) \)

\( i = 1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor} \) but \( i \) value of \( j \)

\( 1 + 2 + 4 + \ldots + 2^{\lfloor \log n \rfloor} = O(n) \)
Loops

```python
i = 1
while i ≤ n {
    for j = 1 to n {
        tap-fingers()
    }
    i = 2*i
}

i = 1
while i ≤ n {
    for j = 1 to i {
        tap-fingers()
    }
    i = 2*i
}
```

### Analysis

- $i = 1, 2, 4, \ldots, 2^\left\lfloor \log n \right\rfloor$ (j=1,2,...,n always)

\[ O(n \log n) \]

### Optimization
- $i = 1, 2, 4, \ldots, 2^\left\lfloor \log n \right\rfloor$ but i value of j

\[ 1 + 2 + 4 + \ldots + 2^\left\lfloor \log n \right\rfloor = O(n) \]

Number of nodes in a complete & full binary tree with (about) n leaves
Recursion
Recursion

Given an array L, find max among numbers between position start and end (inclusive)

```java
findmax (L, start, end) {
    if (start == end) {
        return  L[start]
    } else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```
Recursion

Given an array $L$, find max among numbers between position $start$ and $end$ (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)  
        y = findmax(L,mid+1,end)  
        if (x>y) return x
        else return y
    }
}
```
Recursion

Given an array L, find max among numbers between position start and end (inclusive)

\[
\text{findmax (L, start, end) } \{ \\
\quad \text{if (start == end) } \\
\quad \quad \text{return } L[\text{start}] \\
\quad \text{else } \{ \\
\quad \quad \text{mid} = \lfloor (\text{start+end})/2 \rfloor \\
\quad \quad \text{x = findmax(L,start,mid)} \\
\quad \quad \text{y = findmax(L,mid+1,end)} \\
\quad \quad \text{if (x>y) return x} \\
\quad \quad \text{else return y} \\
\quad \} \\
\}
\]

Time T(n) taken by findmax(L,a,a+n-1)?
\[T(1) = c_1\]
Recursion

Given an array L, find max among numbers between position start and end (inclusive)

```c
findmax (L, start, end) {
    if (start == end)
        return L[start]
    else {
        mid = \lfloor (start+end)/2 \rfloor
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by findmax(L,a,a+n-1)?

\[ T(1) = c_1 \]
\[ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2 \]
Recursion

Given an array L, find max among numbers between position start and end (inclusive)

\[
\text{findmax} \ (L, \text{start}, \text{end}) \ {\begin{cases} 
\text{if} \ (\text{start} == \text{end}) \\
\text{return} \ L[\text{start}] \\
\text{else} \ {\begin{cases} 
\text{mid} = \lfloor (\text{start}+\text{end})/2 \rfloor \\
\text{x} = \text{findmax}(L,\text{start},\text{mid}) \\
\text{y} = \text{findmax}(L,\text{mid}+1,\text{end}) \\
\text{if} \ (x>y) \ \text{return} \ x \\
\text{else return} \ y 
\end{cases} \end{cases}}
\]

Time \( T(n) \) taken by \( \text{findmax}(L,a,a+n-1) \)?

\[
T(1) = c_1 \\
T(n) = T( \lfloor n/2 \rfloor ) + T( \lceil n/2 \rceil ) + c_2
\]

Binary recursion tree with \( c_2 \) on each internal node. \( c_1 \) at leaves.
Recursion

Given an array L, find max among numbers between position start and end (inclusive)

```c
findmax (L, start, end) {
  if (start == end)
    return L[start]
  else {
    mid = ⌊(start+end)/2⌋
    x = findmax(L,start,mid)
    y = findmax(L,mid+1,end)
    if (x>y) return x
    else return y
  }
}
```

Time T(n) taken by
findmax(L,a,a+n-1)?
T(1) = c₁
T(n) = T( ⌊n/2⌋ ) + T( ⌈n/2⌉ ) + c₂

Binary recursion tree with c₂ on each internal node. c₁ at leaves.
T(n) = O(number of nodes)
Recursion

Given an array L, find max among numbers between position start and end (inclusive)

```java
findmax (L, start, end) {
if (start == end)
    return L[start]
else {
    mid = ⌊(start+end)/2⌋
    x = findmax(L,start,mid)
    y = findmax(L,mid+1,end)
    if (x>y) return x
    else return y
}
}
```

Time T(n) taken by

```
T(1) = c_1
T(n) = T( ⌊n/2⌋ ) + T( ⌈n/2⌉ ) + c_2
```

Binary recursion tree with c_2 on each internal node. c_1 at leaves.

T(n) = \(O(\text{number of nodes})\)

T(n) = \(O(n)\)