Asymptotic Notations

CompSci 230

01/31-02/02/2024

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Prelude

- Everything about asymptotic notation in CS230 is the CS-contexted version. *What does this mean?*
- In CS contexts, we are only interested in functions that are:
  - positive;
  - monotonic non-decreasing;
  - (usually) larger than 1.
- As a result, our definitions (of asymptotic notations), as well as some theorem/lemmas, may only work for positive, monotonic non-decreasing, and larger-than-1 functions.
  - Other fields such as math/physics have more nuanced definitions/theorems that extend the scope to functions without those properties.

Asymptotic Notation - Big-O

Definition

$f(n) = O(g(n))$ if $\exists c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \big[f(n) \leq c \cdot g(n) \big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by $c \cdot g(n)$,
  where c is just some (positive) constant.

- Intuitively, $f(n)$ grows at most as fast as $g(n)$
  (omitting constant factors).

Asymptotic Notation - Big-O

Definition

$f(n) = O(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \leq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by $c \cdot g(n)$,
  where c is just some (positive) constant.

- Intuitively, $f(n)$ grows at most as fast as $g(n)$
  (omitting constant factors).

Asymptotic Notation - Big-O

Definition

$f(n) = O(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \leq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by $c \cdot g(n)$,
  where c is just some (positive) constant.

- Intuitively, $f(n)$ grows at most as fast as $g(n)$
  (omitting constant factors).

Asymptotic Notation - Big-O

Definition

$f(n) = O(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \leq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by $c \cdot g(n)$,
  where c is just some (positive) constant.

- Intuitively, $f(n)$ grows at most as fast as $g(n)$
  (omitting constant factors).

Asymptotic Notation - Big-O

Definition

$f(n) = O(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \leq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at most as fast as $g(n)$
  (omitting constant factors).

Asymptotic Notation - Big-O

Definition

$f(n) = O(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \leq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at most as fast as $g(n)$
  (omitting constant factors).

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \left[f(n) \geq c \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \left[f(n) \geq c \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \geq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \left[f(n) \geq c \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \big[f(n) \geq c \cdot g(n)\big]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Ω

Definition

$f(n) = \Omega(g(n))$ if $\exists c > 0\ \exists n_0\ \forall n \geq n_0\ \left[f(n) \geq c \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c \cdot g(n)$,
  where c is just some (positive) constant.
- Intuitively, $f(n)$ grows at least as fast as $g(n)$
  (omitting constant factors).
- $(f(n) = O(g(n))) \iff (g(n) = \Omega(f(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - Big-Θ

Definition

$f(n) = \Theta(g(n))$ if

$\exists c_1 > 0, c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by $c_1 \cdot g(n)$, and above by $c_2 \cdot g(n)$,
  where c_1, c_2 are just some (positive) constants.
- Intuitively, $f(n)$ grows roughly as fast as $g(n)$.
- $(f(n) = \Theta(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-o

Definition

$f(n) = o(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \leq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded above by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly slower than $g(n)$
  (by more than a constant factor).
- $(f(n) = o(g(n))) \iff (f(n) = O(g(n)) \wedge f(n) \neq \Omega(g(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0\ \exists n_0\ \forall n \geq n_0\ \left[f(n) \geq c \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0\ \exists n_0\ \forall n \geq n_0\ \left[f(n) \geq c \cdot g(n)\right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Asymptotic Notations - small-ω (Optional)

Definition

$f(n) = \omega(g(n))$ if $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; \left[f(n) \geq c \cdot g(n) \right]$.

- Eventually (for large enough n),
  $f(n)$ is bounded below by any positive constant times $g(n)$.
- Intuitively, $f(n)$ grows strictly faster than $g(n)$
  (by more than a constant factor).
- $(f(n) = \omega(g(n))) \iff (f(n) = \Omega(g(n)) \wedge f(n) \neq O(g(n)))$.
- $(f(n) = \omega(g(n))) \iff (g(n) = o(f(n)))$.

Limit-Based Definition

- You might have seen in CS201 that $f(n) = O(g(n))$ if $\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$
- That works *most of the time*, but that definition is not equivalent to ours:
  $f(n) = O(g(n))$ if $\exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : \; f(n) \leq c \cdot g(n)$
  - The problem is the limit might not necessarily exist.
  - Example: $f(n) = \left(3 + (-1)^n\right) \cdot n$, $g(n) = n$
- The "correct" limit-based definition is
  $f(n) = O(g(n))$ if $\limsup_{x \to \infty} \frac{|f(x)|}{g(x)} < \infty$
  - We won't use it.

Limit-Based Definition

- You might have seen in CS201 that
  $f(n) = O(g(n))$ if $\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$
- That works *most of the time*, but that definition is not equivalent to ours:
  $f(n) = O(g(n))$ if $\exists c > 0 \ \exists n_0 \ \forall n \geq n_0 : \ f(n) \leq c \cdot g(n)$
  - The problem is the limit might not necessarily exist.
  - Example: $f(n) = \left(3 + (-1)^n\right) \cdot n$, $g(n) = n$
- The "correct" limit-based definition is
  $f(n) = O(g(n))$ if $\limsup_{x \to \infty} \frac{|f(x)|}{g(x)} < \infty$
  - We won't use it.

Limit-Based Definition

- You might have seen in CS201 that
  $f(n) = O(g(n))$ if $\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$
- That works *most of the time*, but that definition is not equivalent to ours:
  $f(n) = O(g(n))$ if $\exists c > 0 \ \exists n_0 \ \forall n \geq n_0 : \ f(n) \leq c \cdot g(n)$
  - The problem is the limit might not necessarily exist.
  - Example: $f(n) = \left(3 + (-1)^n\right) \cdot n$, $g(n) = n$
- The "correct" limit-based definition is
  $f(n) = O(g(n))$ if $\limsup_{x \to \infty} \frac{|f(x)|}{g(x)} < \infty$
  - We won't use it.

Limit-Based Definition

- You might have seen in CS201 that
  $f(n) = O(g(n))$ if $\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$
- That works *most of the time*, but that definition is not equivalent to ours:
  $f(n) = O(g(n))$ if $\exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : \; f(n) \leq c \cdot g(n)$
  - The problem is the limit might not necessarily exist.
  - Example: $f(n) = \left(3 + (-1)^n\right) \cdot n$, $g(n) = n$
- The "correct" limit-based definition is
  $f(n) = O(g(n))$ if $\limsup_{x \to \infty} \frac{|f(x)|}{g(x)} < \infty$
  - We won't use it.

More high-level understanding of all 5 notations

Table: Sloppy but highly useful mental images.

$f(n) = O(g(n))$	$f \leq g$
$f(n) = \Omega(g(n))$	$f \geq g$
$f(n) = \Theta(g(n))$	$f = g$
$f(n) = o(g(n))$	$f < g$
$f(n) = \omega(g(n))$	$f > g$

Peer Instruction: Asymptotic notation 1

Peer Instruction: Asymptotic notation 2

Peer Instruction: Asymptotic notation 2

- $f(n) = O(g(n)) \implies 2^{f(n)} = O(2^{g(n)})$
- $f(n) = O(g(n)) \implies \ln f(n) = O(\ln g(n))$

Peer Instruction: Asymptotic notation 2

- $f(n) = O(g(n)) \implies 2^{f(n)} = O(2^{g(n)})$
  Counterexample: let $f(n) = 2n$, and $g(n) = n$.
  Then $2^{f(n)} = 2^{2n} = (2^n)^2 = \omega(2^n) = \omega(g(n))$.

- $f(n) = O(g(n)) \implies \ln f(n) = O(\ln g(n))$

Peer Instruction: Asymptotic notation 2

- $f(n) = O(g(n)) \implies 2^{f(n)} = O(2^{g(n)})$
  Counterexample: let $f(n) = 2n$, and $g(n) = n$.
  Then $2^{f(n)} = 2^{2n} = (2^n)^2 = \omega(2^n) = \omega(g(n))$.

- $f(n) = O(g(n)) \implies \ln f(n) = O(\ln g(n))$
  This is true (given the assumptions).
  You will prove it in recitations.

Beware of adding $O(1)$

- Claim: $1 + 2 + 3 + \ldots + n = O(n)$.

Beware of adding $O(1)$

- Claim: $1 + 2 + 3 + \ldots + n = O(n)$.
  Proof: because each term i is $O(1)$, and there are n terms, the sum must be $O(n)$.

Beware of adding $O(1)$

- Claim: $1 + 2 + 3 + \ldots + n = O(n)$.
  Proof: because each term i is $O(1)$, and there are n terms, the sum must be $O(n)$.

- i depends on n and is thus NOT a constant.

Solving Recurrence Relations

Asymptotic notations are useful for solving
(and describing the behavior of) recurrence relations, since:

- They often do not sum up to a closed form.
- Or, it makes little sense for the recurrence relation itself to capture the constants!
  - Take the `MergeSort` algorithm as an example:
    $T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$
    (refer to your CS201 material for implementation details)
- Our goal is now to be able to solve the recurrence relation for `MergeSort` and get $T(n) = O(n \log n)$.

Solving Recurrence Relations

Asymptotic notations are useful for solving
(and describing the behavior of) recurrence relations, since:

- They often do not sum up to a closed form.
- Or, it makes little sense for the recurrence relation itself to capture the constants!
  - Take the `MergeSort` algorithm as an example:
    $T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$
    (refer to your CS201 material for implementation details)
- Our goal is now to be able to solve the recurrence relation for `MergeSort` and get $T(n) = O(n \log n)$.

Solving Recurrence Relations

Asymptotic notations are useful for solving
(and describing the behavior of) recurrence relations, since:

- They often do not sum up to a closed form.
- Or, it makes little sense for the recurrence relation itself to capture the constants!
  - Take the `MergeSort` algorithm as an example:
    $T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$
    (refer to your CS201 material for implementation details)
- Our goal is now to be able to solve the recurrence relation for `MergeSort` and get $T(n) = O(n \log n)$.

recurrence relations and expectations in 201

- In general, will **not** be asked to **solve** recurrence relations on exams (for later classes in theory).
- You **may** be asked to determine the recurrence relation of a given algorithm/code.

Recurrence	Algorithm	Solution
`T(n) = T(n/2) + O(1)`	binary search	`O(log n)`
`T(n) = T(n-1) + O(1)`	sequential search	`O(n)`
`T(n) = 2T(n/2) + O(1)`	tree traversal	`O(n)`
`T(n) = T(n/2) + O(n)`	qsort partition ,find k^{th}	`O(n)`
`T(n) = 2T(n/2) + O(n)`	mergesort, quicksort	`O(n log n)`
`T(n) = T(n-1) + O(n)`	selection or bubble sort	$O(n^2)$

Solving Recurrence Relations

- Method I: Guess-and-Verify

  Work out the first few terms, find the pattern, then verify

- Method II: Substitution

  Repeatedly apply the recurrence until reaching the base case(s)

- Method III: Tree method

  Draw out a conceptual tree of the recurrence; then estimate (i) the tree height, and (ii) the partial sum within one layer

- Method IV: Master's Theorem (optional)

Solving Recurrence Relations

- Method I: Guess-and-Verify

  Work out the first few terms, find the pattern, then verify

- Method II: Substitution

  Repeatedly apply the recurrence until reaching the base case(s)

- Method III: Tree method

  Draw out a conceptual tree of the recurrence; then estimate (i) the tree height, and (ii) the partial sum within one layer

- Method IV: Master's Theorem (optional)

Solving Recurrence Relations

- Method I: Guess-and-Verify

  Work out the first few terms, find the pattern, then verify

- Method II: Substitution

  Repeatedly apply the recurrence until reaching the base case(s)

- Method III: Tree method

  Draw out a conceptual tree of the recurrence; then estimate (i) the tree height, and (ii) the partial sum within one layer

- Method IV: Master's Theorem (optional)

Solving Recurrence Relations

- Method I: Guess-and-Verify

  Work out the first few terms, find the pattern, then verify

- Method II: Substitution

  Repeatedly apply the recurrence until reaching the base case(s)

- Method III: Tree method

  Draw out a conceptual tree of the recurrence; then estimate (i) the tree height, and (ii) the partial sum within one layer

- Method IV: Master's Theorem (optional)

Solving Recurrence Relations

- Method I: Guess-and-Verify

  Work out the first few terms, find the pattern, then verify

- Method II: Substitution

  Repeatedly apply the recurrence until reaching the base case(s)

- Method III: Tree method

  Draw out a conceptual tree of the recurrence; then estimate (i) the tree height, and (ii) the partial sum within one layer

- Method IV: Master's Theorem (optional)

  Again, *optional* means not required *but allowed* on assignments/exams.

Substitution - `MergeSort`

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

$$T(n) = 2T(\frac{n}{2}) + cn$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

$$T(n) = 2T(\frac{n}{2}) + cn$$
$$= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn \\
&= 4T(\frac{n}{4}) + cn + cn
\end{aligned}
$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn \\
&= 4T(\frac{n}{4}) + cn + cn \\
&= 4\left(2T(\frac{n}{8}) + \frac{cn}{4}\right) + cn + cn
\end{aligned}
$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn \\
&= 4T(\frac{n}{4}) + cn + cn \\
&= 4\left(2T(\frac{n}{8}) + \frac{cn}{4}\right) + cn + cn \\
&= 8T(\frac{n}{8}) + cn + cn + cn \\
&\vdots
\end{aligned}
$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn, T(1) = 1,$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn \\
&= 4T(\frac{n}{4}) + cn + cn \\
&= 4\left(2T(\frac{n}{8}) + \frac{cn}{4}\right) + cn + cn \\
&= 8T(\frac{n}{8}) + cn + cn + cn \\
&\vdots \\
&= nT(1) + \log n \cdot cn
\end{aligned}
$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn$, $T(1) = 1$,

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn \\
&= 4T(\frac{n}{4}) + cn + cn \\
&= 4\left(2T(\frac{n}{8}) + \frac{cn}{4}\right) + cn + cn \\
&= 8T(\frac{n}{8}) + cn + cn + cn \\
&\vdots \\
&= nT(1) + \log n \cdot cn \\
&= n + c(n \log n) = O(n \log n).
\end{aligned}
$$

Substitution - MergeSort

$T(n) = 2T(\frac{n}{2}) + cn$, $T(1) = 1$, cn comes from $O(n)$ in merging.

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn \\
&= 2\left(2T(\frac{n}{4}) + \frac{cn}{2}\right) + cn \\
&= 4T(\frac{n}{4}) + cn + cn \\
&= 4\left(2T(\frac{n}{8}) + \frac{cn}{4}\right) + cn + cn \\
&= 8T(\frac{n}{8}) + cn + cn + cn \\
&\vdots \\
&= nT(1) + \log n \cdot cn \\
&= n + c(n \log n) = O(n \log n).
\end{aligned}
$$

Tree method - BinarySearch

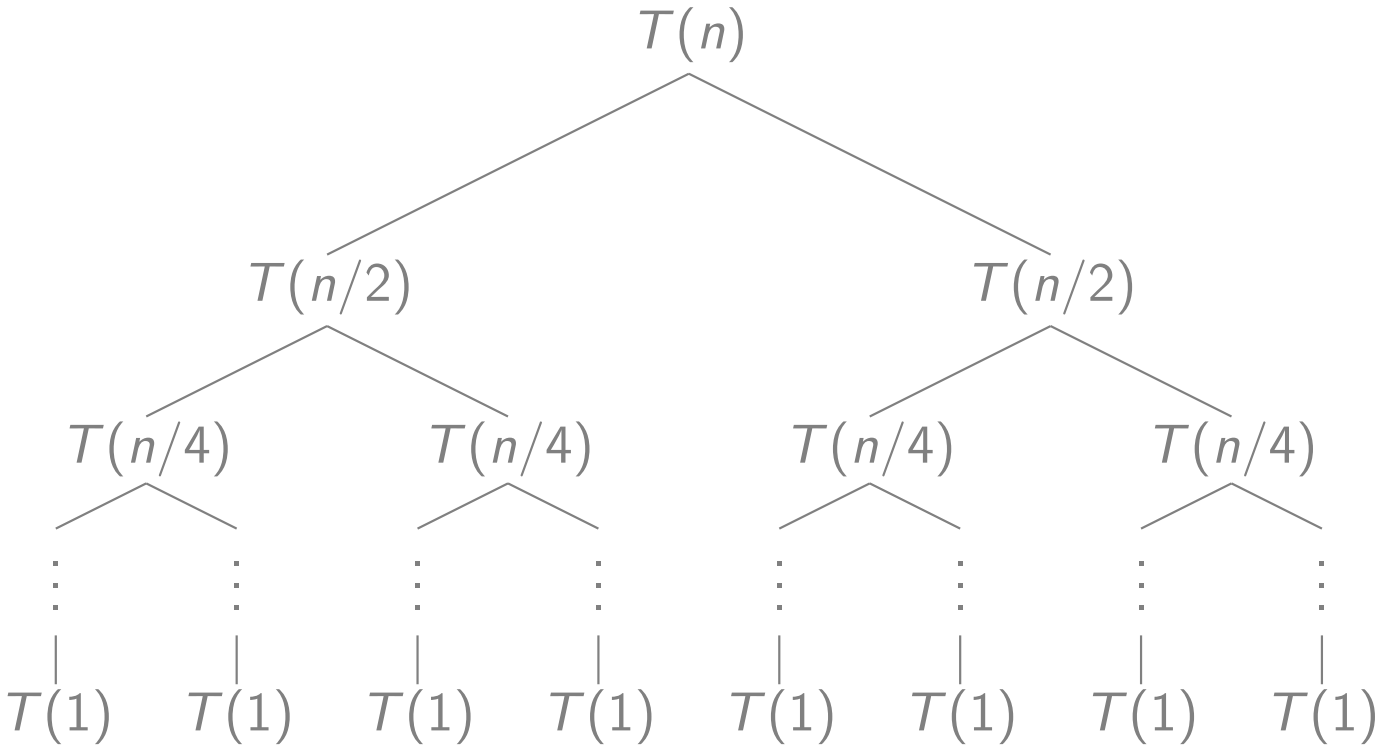
Algorithm Binary Search (A, n, T)

Input: Array $A[1 \ldots n]$; element T
Output: m: index of T in A

1: $L \leftarrow 0$
2: $R \leftarrow n - 1$
3: **while** $L \leq R$ **do**
4: $\quad m \leftarrow \lceil \frac{L+R}{2} \rceil$
5: $\quad$ **if** $A[m] < T$ **then**
6: $\quad\quad L \leftarrow m + 1$
7: $\quad$ **else if** $A[m] > T$ **then**
8: $\quad\quad R \leftarrow m - 1$
9: $\quad$ **else**
10: $\quad\quad$ **return** m
11: $\quad$ **end if**
12: **end while**

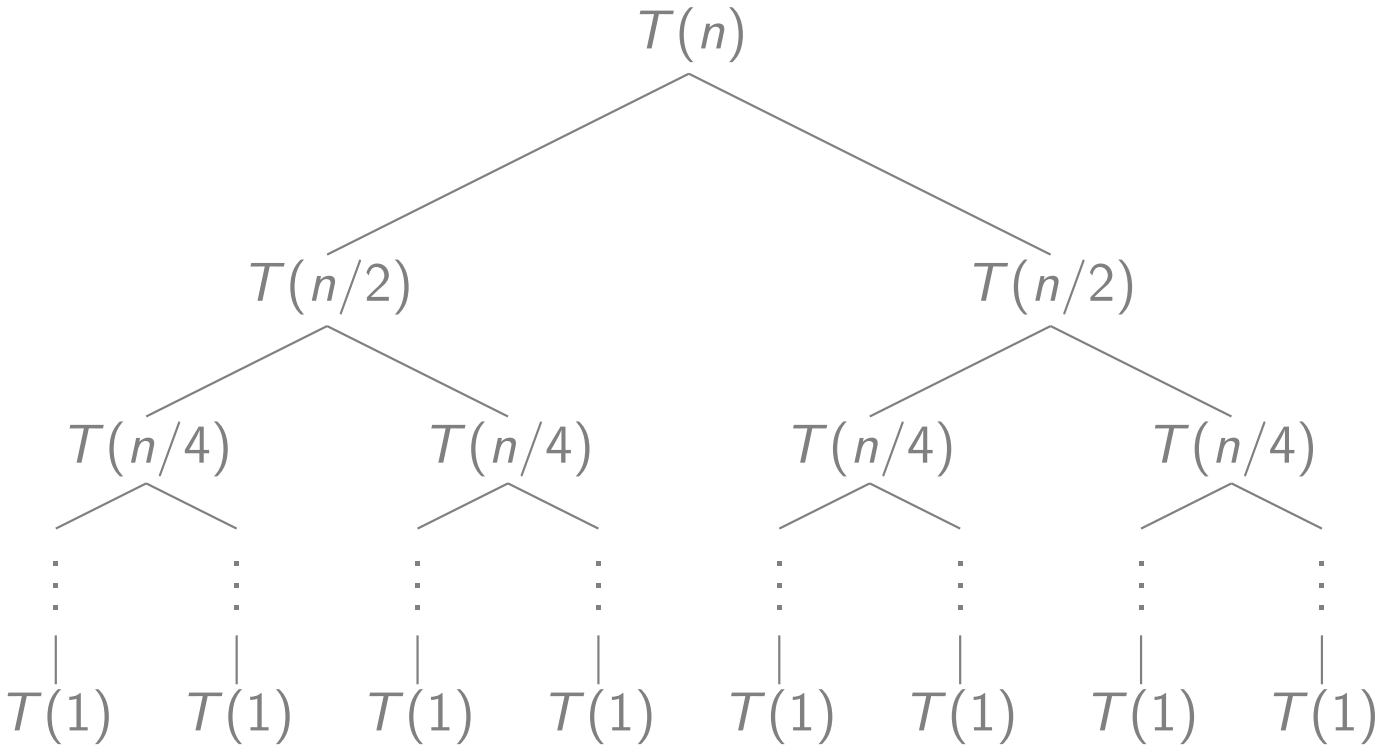
Tree method - `BinarySearch`

$T(n) = T(\frac{n}{2}) + O(1), T(1) = 1$



Tree method - `BinarySearch`

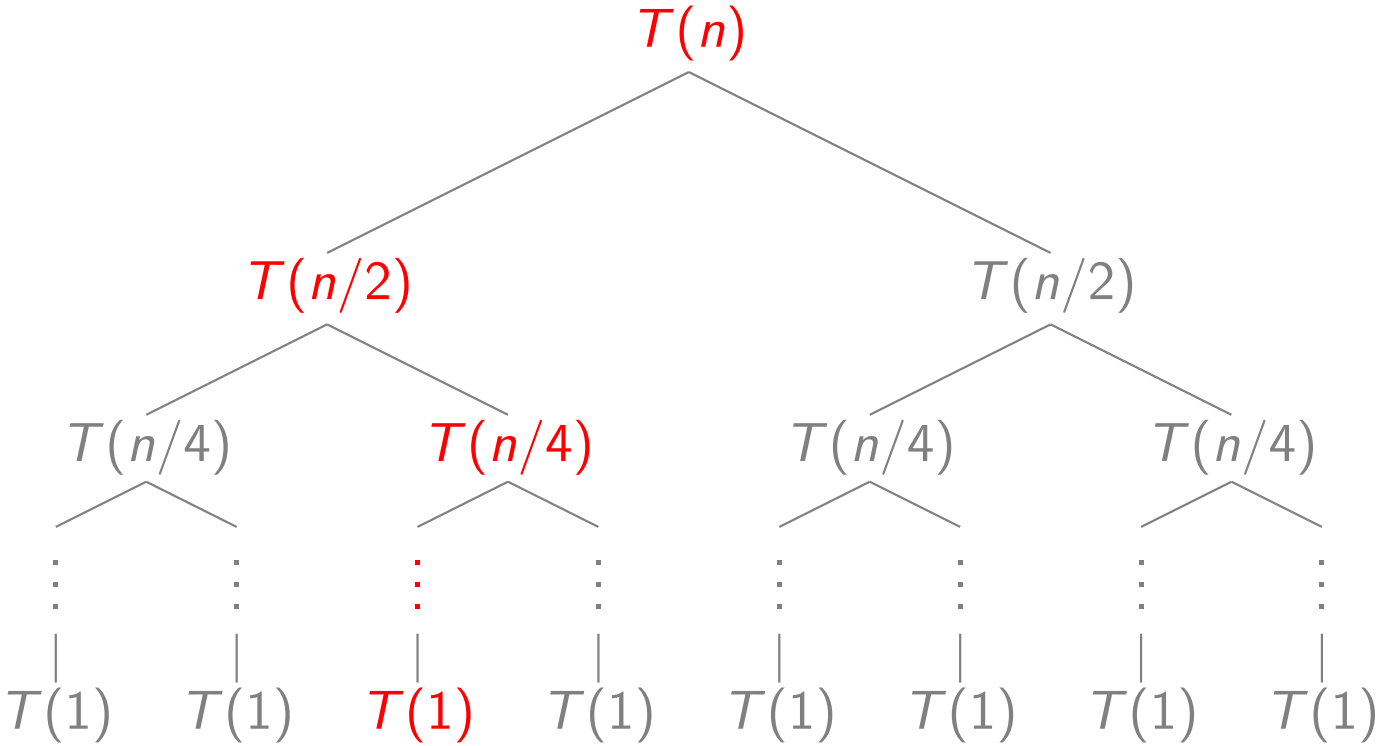
$T(n) = T(\frac{n}{2}) + O(1), T(1) = 1$



Does every node in this recursion tree happen?

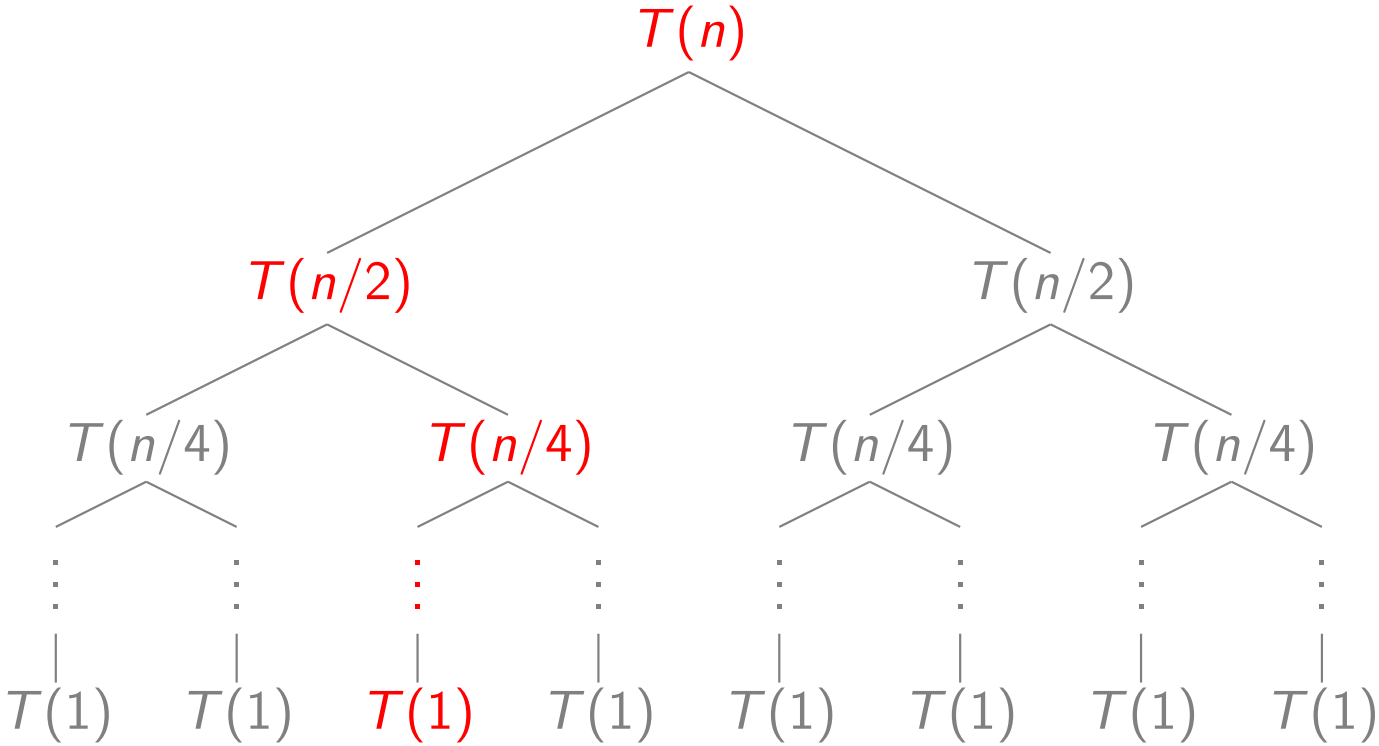
Tree method - `BinarySearch`

$T(n) = T(\frac{n}{2}) + O(1), T(1) = 1$



Tree method - BinarySearch

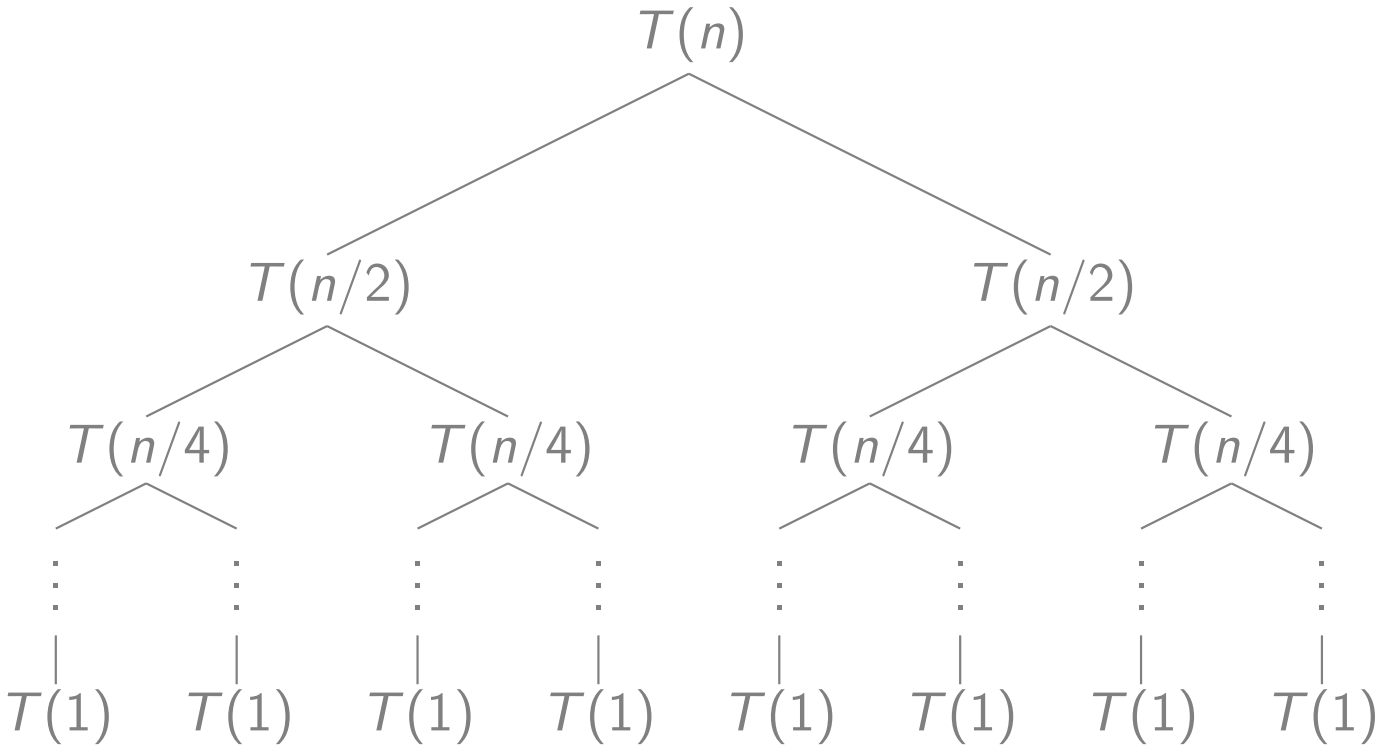
$T(n) = T(\frac{n}{2}) + O(1), T(1) = 1$



$\implies T(n) = O(\log n) \cdot O(1) = O(\log n)$

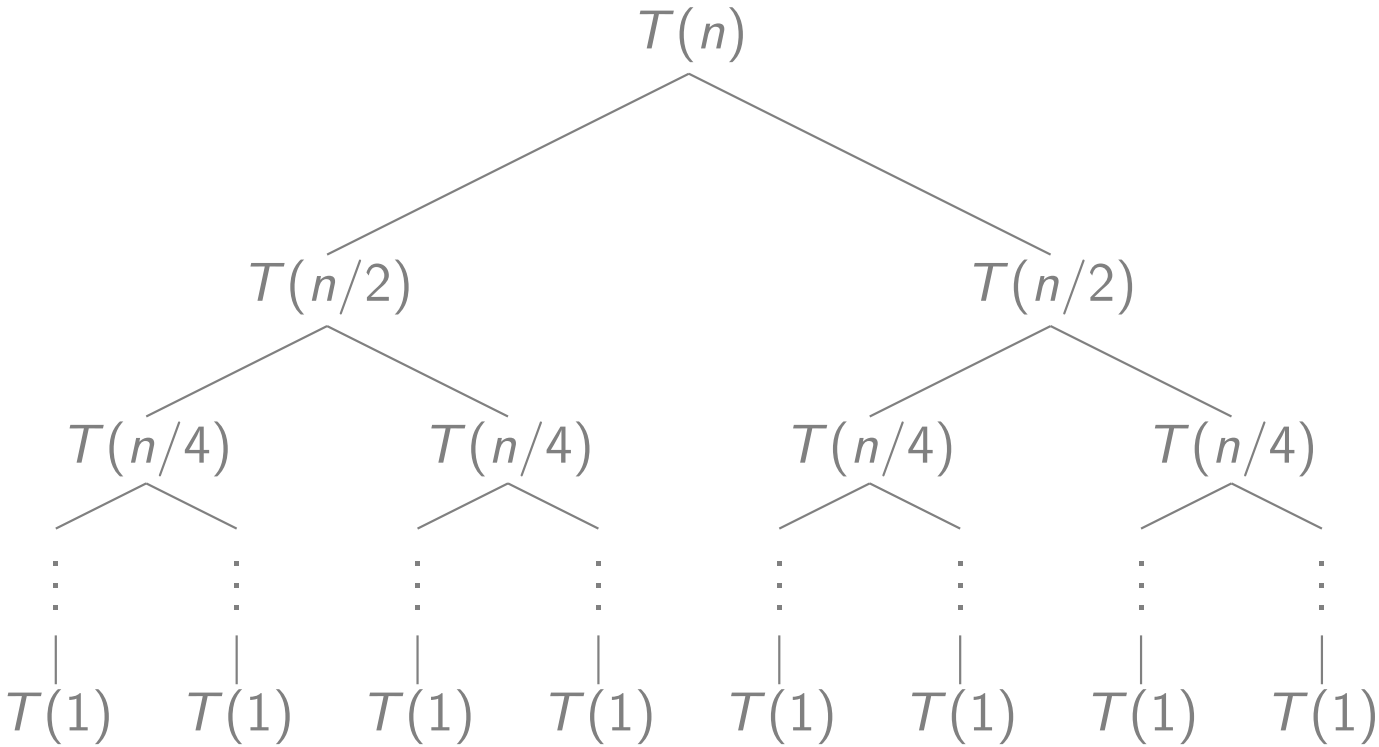
Tree method - `MergeSort`

$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



Tree method - MergeSort

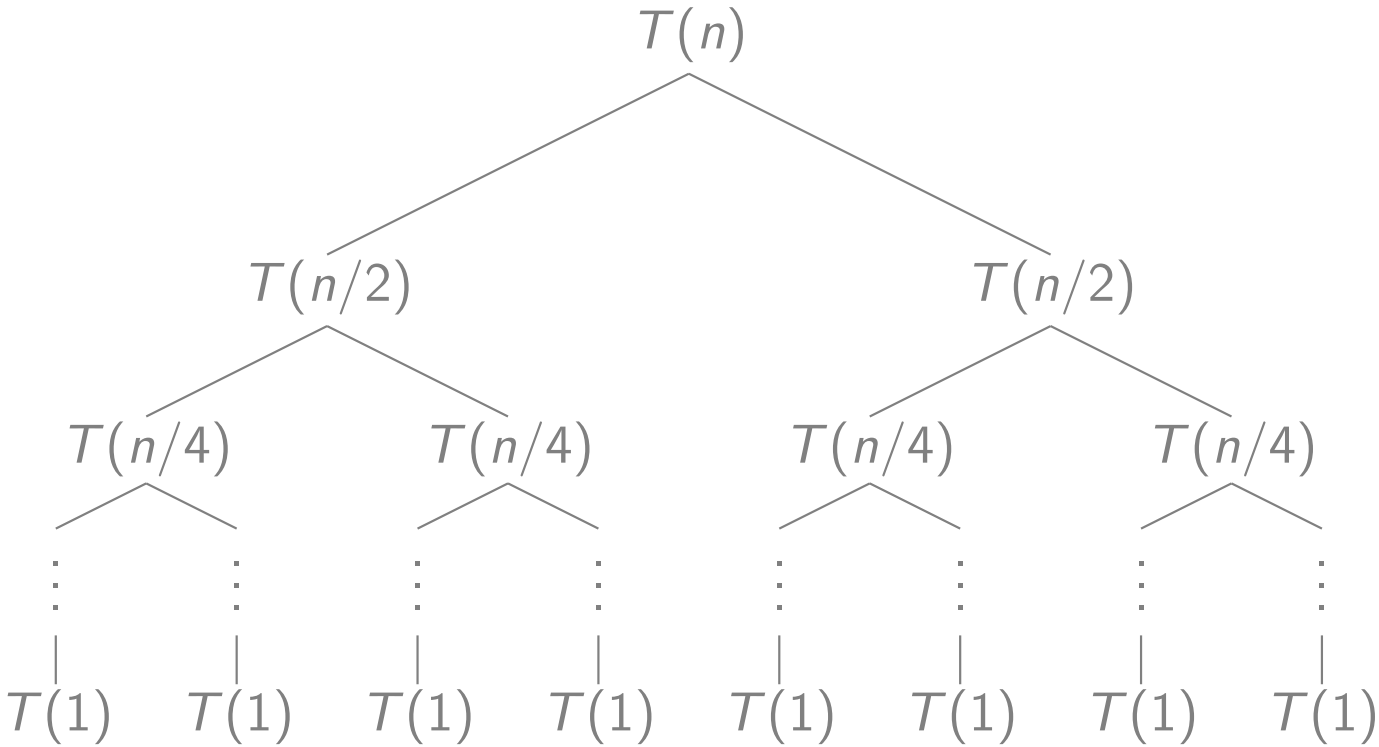
$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



Does every node in this recursion tree happen?

Tree method - MergeSort

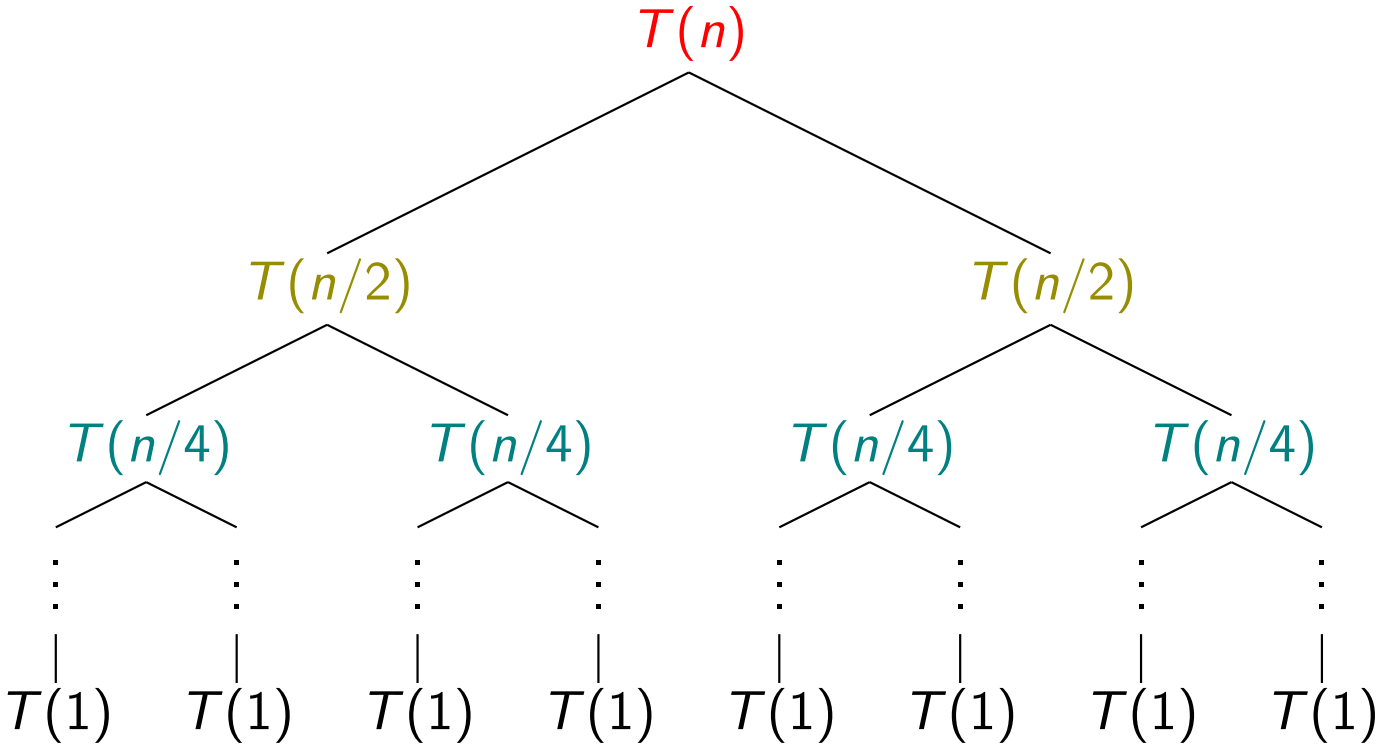
$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



Does every node in this recursion tree happen? Yes it does.

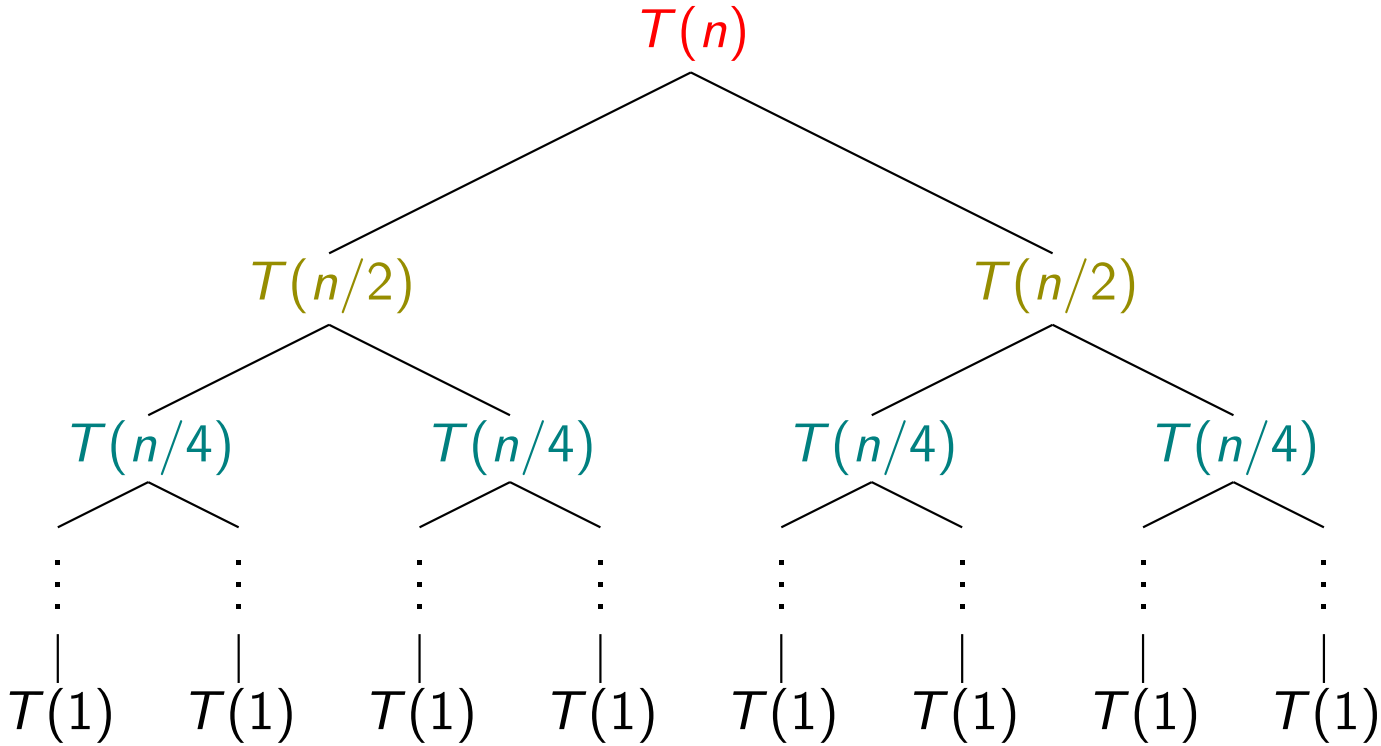
Tree method - MergeSort

$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



Tree method - MergeSort

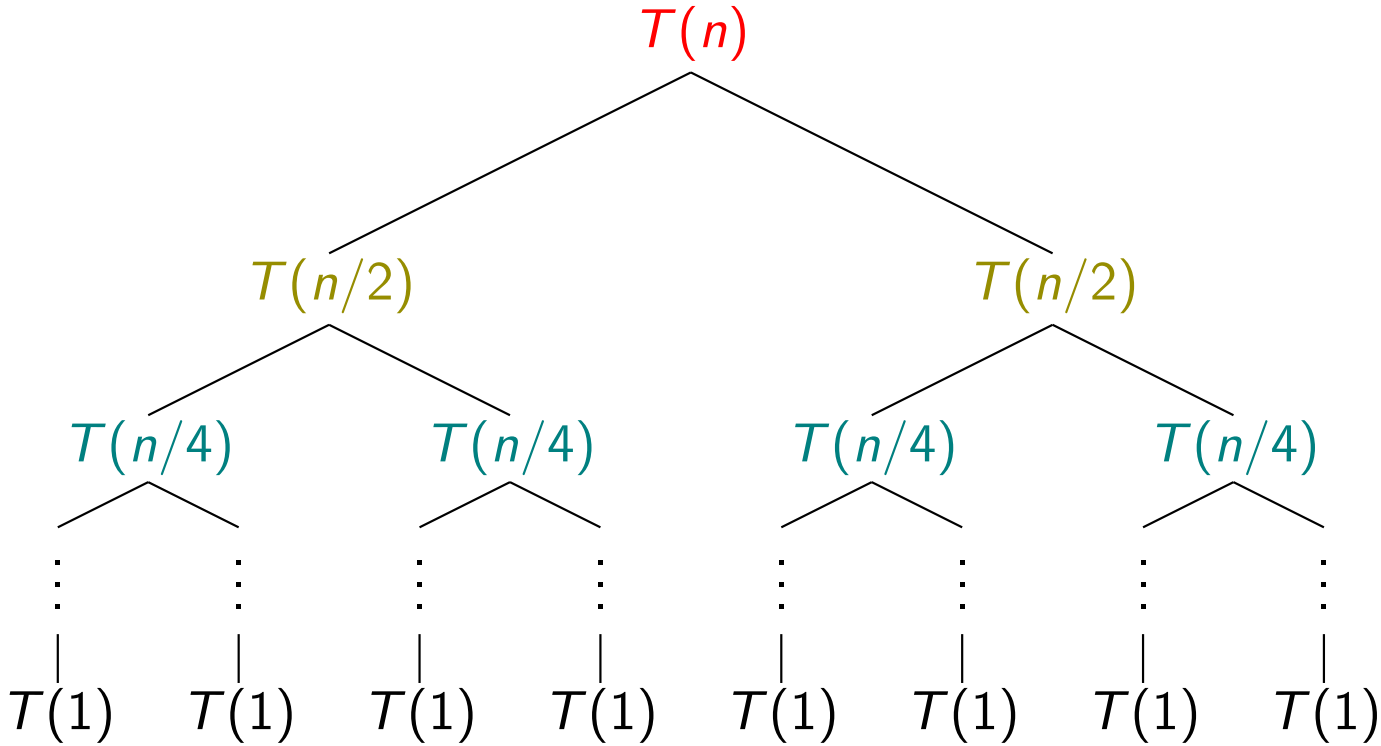
$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



Root level: $O(n)$

Tree method - MergeSort

$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$

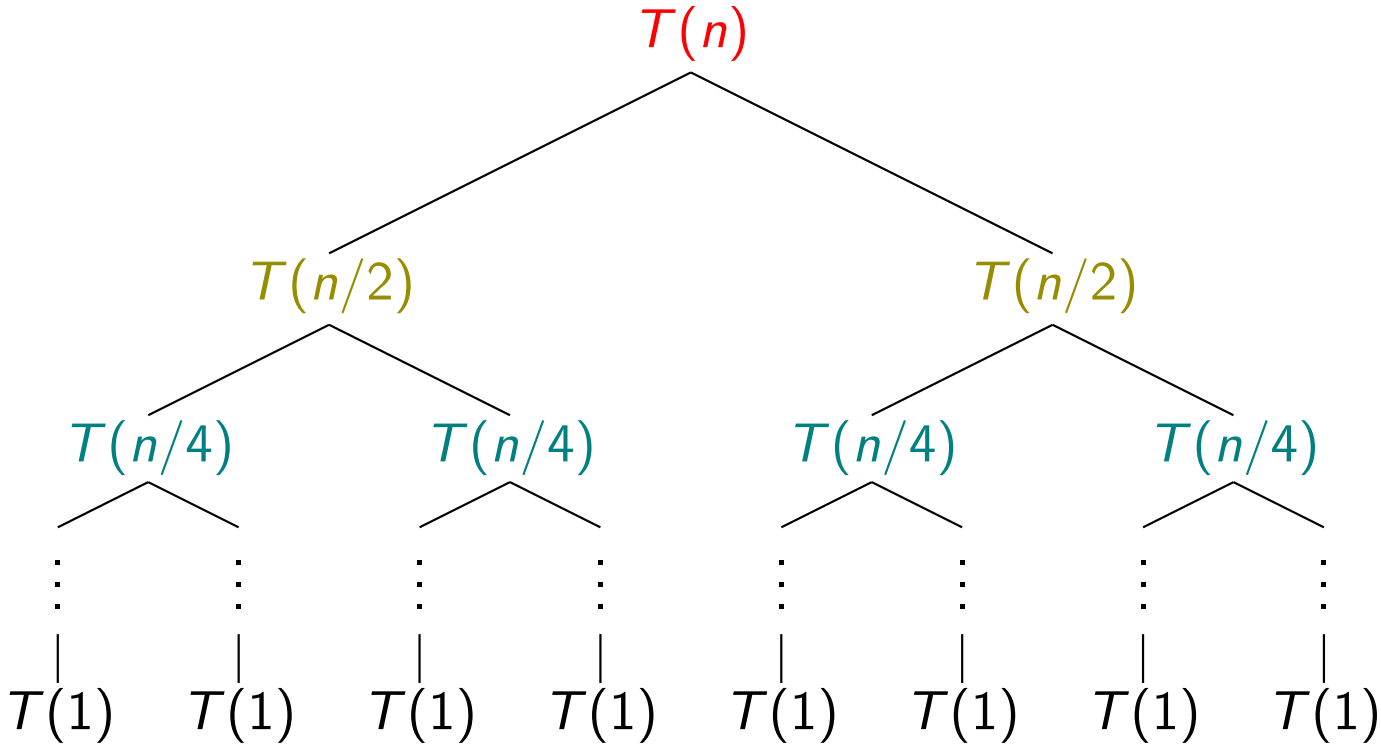


Root level: $O(n)$

First level: $2 \times O(n/2) = O(n)$

Tree method - MergeSort

$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



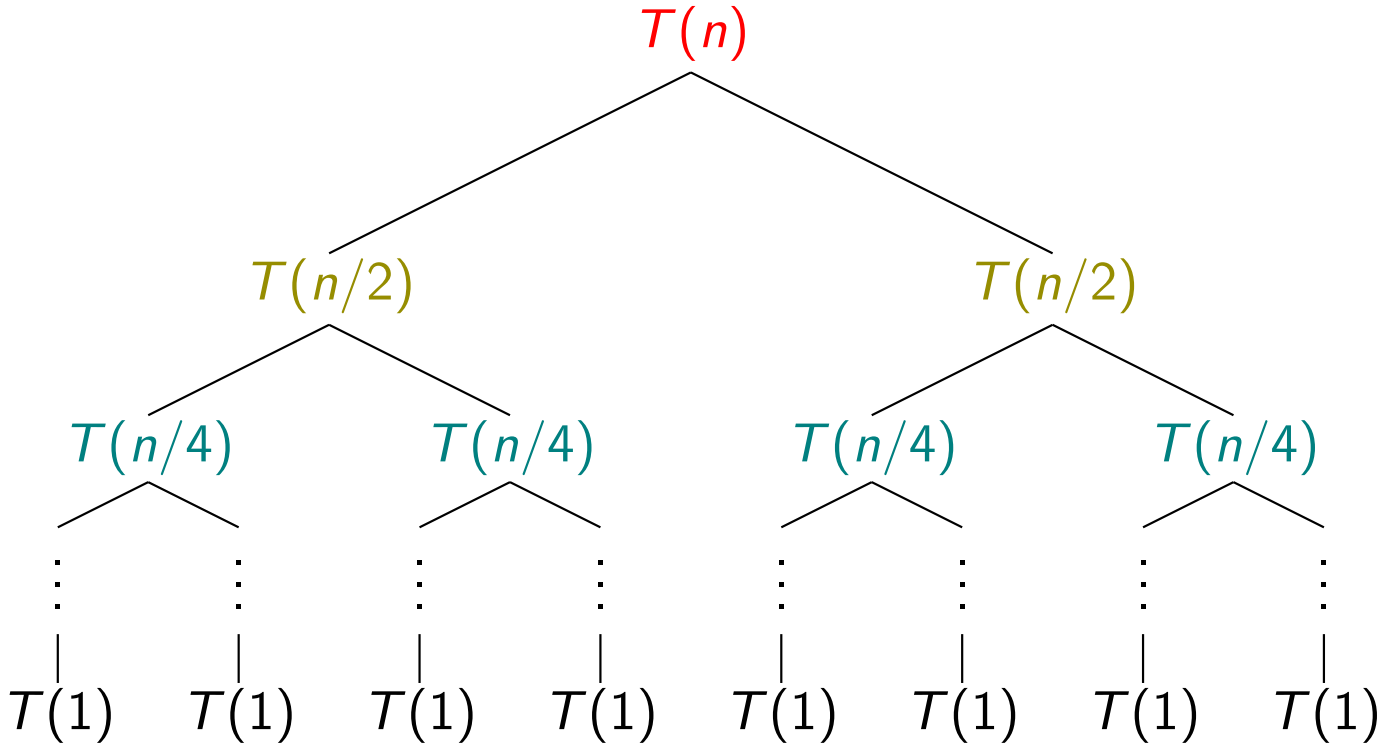
Root level: $O(n)$

First level: $2 \times O(n/2) = O(n)$

Second level: $4 \times O(n/4) = O(n)$

Tree method - MergeSort

$T(n) = 2T(\frac{n}{2}) + O(n), T(1) = 1$



Root level: $O(n)$

First level: $2 \times O(n/2) = O(n)$

Second level: $4 \times O(n/4) = O(n)$

... every level is $O(n)$ and there are $O(\log n)$ levels,
so the total work is $O(n \log n)$.

What does the $=$ sign mean in asymptotic notations?

- Claim: $1 = 2$.

What does the $=$ sign mean in asymptotic notations?

- Claim: $1 = 2$.
  Proof: $n = O(n)$. $2n = O(n)$.
  Therefore, $n = O(n) = 2n$, so $1 = 2$.

What does the $=$ sign mean in asymptotic notations?

- Claim: $1 = 2$.
  Proof: $n = O(n)$. $2n = O(n)$.
  Therefore, $n = O(n) = 2n$, so $1 = 2$.

- Blame the first computer scientist that wrote $f(n) = O(g(n))$.

What does the $=$ sign mean in asymptotic notations?

- Claim: $1 = 2$.
  Proof: $n = O(n)$. $2n = O(n)$.
  Therefore, $n = O(n) = 2n$, so $1 = 2$.

- Blame the first computer scientist that wrote $f(n) = O(g(n))$.

- We should have written $f(n) \in O(g(n))$,
  to represent that $O(g(n))$ is a *set* or *class* of functions.

What does the $=$ sign mean in asymptotic notations?

- Claim: $1 = 2$.
  Proof: $n = O(n)$. $2n = O(n)$.
  Therefore, $n = O(n) = 2n$, so $1 = 2$.

- Blame the first computer scientist that wrote $f(n) = O(g(n))$.

- We should have written $f(n) \in O(g(n))$,
  to represent that $O(g(n))$ is a *set* or *class* of functions.
  - This way nothing goes wrong when we say $n \in O(n)$ and $2n \in O(n)$.

More Misuses of Asymptotic Notations

Do not say:

"*The running time of Algorithm X is at least* $O(n^2)$".

More Misuses of Asymptotic Notations

Do not say:

"*The running time of Algorithm X is at least* $O(n^2)$".

- Big-O is only an upper bound!

More Misuses of Asymptotic Notations

Do not say:
"*The running time of Algorithm X is at least* $O(n^2)$".

- Big-O is only an upper bound!
- A constant is also $O(n^2)$;
  therefore, the above sentence is meaningless.

More Misuses of Asymptotic Notations

Do not say:
"*The running time of Algorithm X is at least* $O(n^2)$".

- Big-O is only an upper bound!
- A constant is also $O(n^2)$;
  therefore, the above sentence is meaningless.

- Consider:
  "*The running time of Algorithm X is at least* $\Theta(n^2)$".

More Misuses of Asymptotic Notations

Do not say:
"*The running time of Algorithm X is at least* $O(n^2)$".

- Big-O is only an upper bound!
- A constant is also $O(n^2)$;
  therefore, the above sentence is meaningless.

- Consider:
  "*The running time of Algorithm X is at least* $\Theta(n^2)$".
- Or even better:
  "*The running time of Algorithm X is* $\Omega(n^2)$".

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \\ \Theta(n^d \log n) & \text{if } d = c \\ \Theta(n^d) & \text{if } d > c \end{cases}$$

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \quad \text{“leaf-heavy”} \\ \Theta(n^d \log n) & \text{if } d = c \\ \Theta(n^d) & \text{if } d > c \end{cases}$$

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \quad \text{“leaf-heavy”} \\ \Theta(n^d \log n) & \text{if } d = c \\ \Theta(n^d) & \text{if } d > c \quad \text{“root-heavy”} \end{cases}$$

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \quad \text{“leaf-heavy”} \\ \Theta(n^d \log n) & \text{if } d = c \quad \text{“balanced”} \\ \Theta(n^d) & \text{if } d > c \quad \text{“root-heavy”} \end{cases}$$

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \quad \text{“leaf-heavy”} \\ \Theta(n^d \log n) & \text{if } d = c \quad \text{“balanced”} \\ \Theta(n^d) & \text{if } d > c \quad \text{“root-heavy”} \end{cases}$$

- Both `BinarySearch` and `MergeSort` are “balanced”.

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \quad \text{“leaf-heavy”} \\ \Theta(n^d \log n) & \text{if } d = c \quad \text{“balanced”} \\ \Theta(n^d) & \text{if } d > c \quad \text{“root-heavy”} \end{cases}$$

- Both `BinarySearch` and `MergeSort` are “balanced”.
- As you can see, this won't solve all recurrences.
  - More generalized forms of Master Theorem exist (out of scope).

Master Theorem (Optional)

- For a recurrence of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, where $a \geq 1, b > 1$ are constants: let $c = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } d < c \quad \text{“leaf-heavy”} \\ \Theta(n^d \log n) & \text{if } d = c \quad \text{“balanced”} \\ \Theta(n^d) & \text{if } d > c \quad \text{“root-heavy”} \end{cases}$$

- Both `BinarySearch` and `MergeSort` are “balanced”.
- As you can see, this won't solve all recurrences.
  - More generalized forms of Master Theorem exist (out of scope).
- Master Theorem can be proved by *induction*.
  - A lot of stuff in CM3 also can, but we save that to CM5.