CS 240E – Data Structures and Data Management (Enriched)

Module 1: Introduction and Asymptotic Analysis

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2026

version *2026-05-12 02:43*

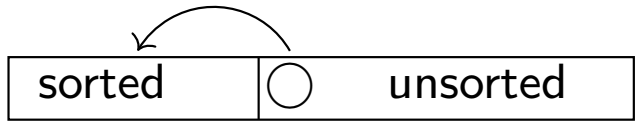
Outline

Outline

Algorithms and problems: Review

Problem: Description of possible input and desired output.
Example: Sorting problem.

Algorithm: *Step-by-step process*, works on any **instance** *I*.
Example: *insertion-sort*



1) Describe the overall idea
"Keep part of array sorted, and repeatedly add more to sorted part."

2) Give **pseudo-code** or detailed description.

```
insertion-sort(A, n)
A: array of size n
1.  for (i ← 1; i < n; i++) do
2.      for (j ← i; j > 0 and A[j−1] > A[j]; j−−) do
3.          swap A[j] and A[j − 1]
```

Pseudo-code: designed for a person, not for a computer.

Algorithms and problems: Review

3) Argue correctness.

- Typically state loop-invariants, or other key ingredients, but no need for a formal (CS245-style) proof by induction.
- Sometimes obvious enough from idea-description and comments.

4) Analyze the algorithm.

- We want to bound the number of **primitive operations**
- We want to bound the **auxiliary space**
- We need a computer model: **Random Access Model** (RAM)
  - unlimited set of memory cells
  - any number fits into a cell (but do not abuse)
  - standard arithmetic operations, but no $\sqrt{\cdot}$, sin, log, ...
  - all operations take the same amount of time
- We do not count exactly, instead use **asymptotic analysis** (big-O)

Outline

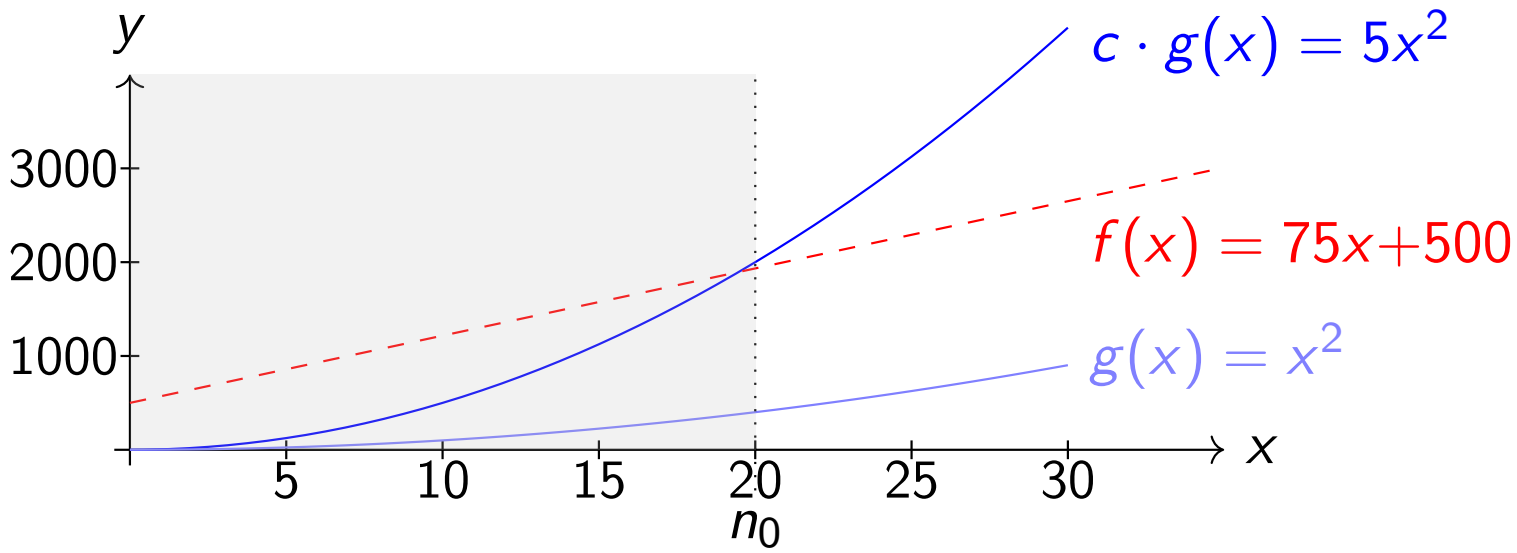
Order notation overview

Symbol and acronyms			picture / definition	Typical use
O	big-O	asymptotic upper bound		*merge-sort* takes $O(n \log n)$ time.
Ω	big-Omega	asymptotic lower bound		*insertion-sort* may take $\Omega(n^2)$ time.
Θ	Theta	asymptotically the same/tight		*insertion-sort* has worst-case run-time $\Theta(n^2)$
o	little-o	asymptotically strictly smaller		*merge-sort* is asymp. faster than *insertion-sort* in worst case.
ω	little-omega	asymptotically strictly bigger		*merge-sort* uses $\omega(1)$ aux. space.

Order notation

Study relationships between *functions*.

Example: $f(x) = 75x + 500$ and $g(x) = x^2$ (e.g. $c = 5, n_0 = 20$)



***O*-notation:** $f(x) \in O(g(x))$ (f is *asymptotically upper-bounded* by g) if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $|f(x)| \leq c\,|g(x)|$ for all $x \geq n_0$.

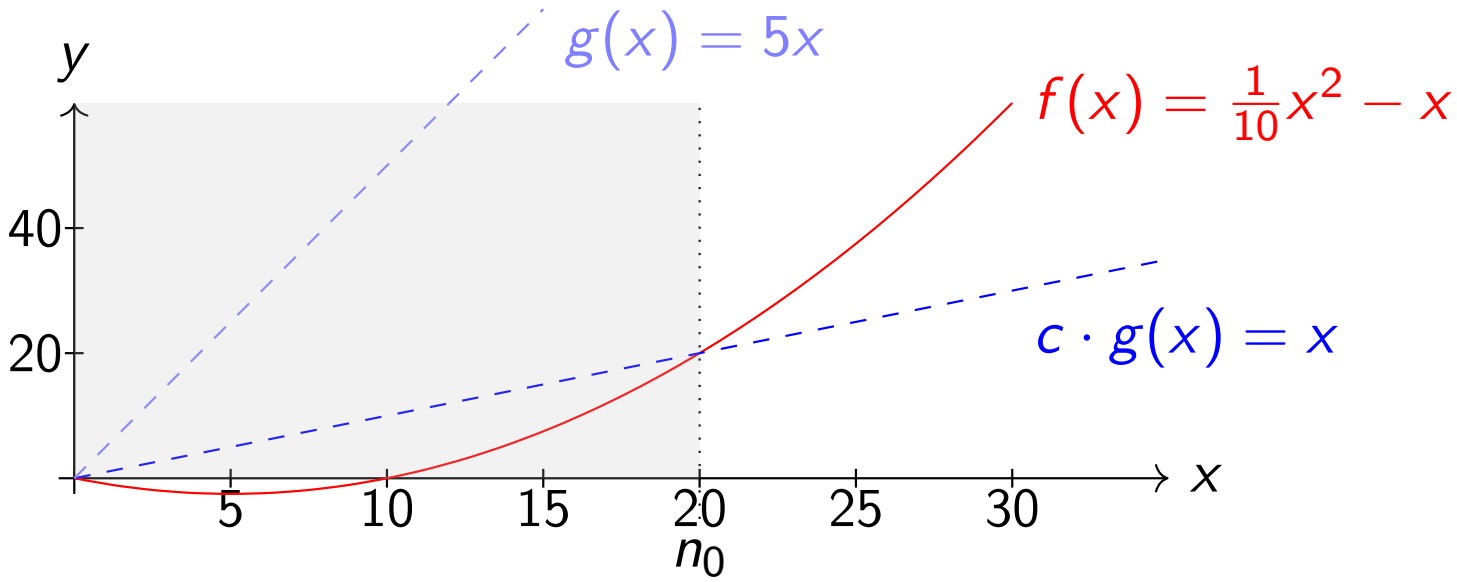
In CS240: Parameter is usually an integer (write n rather than x).
$f(n), g(n)$ usually positive for sufficiently big n (omit absolute value signs).

Asymptotic lower bound

- We have $2n^2 + 3n + 11 \in O(n^2)$.
- But we also have $2n^2 + 3n + 11 \in O(n^{10})$.
- We want a *tight* asymptotic bound.

Ω-notation: $f(x) \in \Omega(g(x))$ (f is *asymptotically lower-bounded* by g) if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $c\,|g(x)| \leq |f(x)|$ for all $x \geq n_0$.

Example: $f(x) = \frac{1}{10}x^2 - x$ and $g(x) = 5x$ (e.g. $c = \frac{1}{5}, n_0 = 20$)



Asymptotic tight bound

Θ-notation: $f(x) \in \Theta(g(x))$ (f is *asymptotically tightly-bounded* by g) if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that

$$c_1 |g(x)| \leq |f(x)| \leq c_2 |g(x)| \text{ for all } x \geq n_0.$$

Equivalently: $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

We also say that *the growth rates of f and g are the same*. Typically, $f(x)$ may be "complicated" and $g(x)$ is chosen to be a very simple function.

Example: Prove that $\log_b(n) \in \Theta(\log n)$ for all $b > 1$ from first principles.

Common growth rates

Commonly encountered growth rates in analysis of algorithms include the following:

- $\Theta(1)$ (*constant*),
- $\Theta(\log n)$ (*logarithmic*),
- $\Theta(n)$ (*linear*),
- $\Theta(n \log n)$ (*linearithmic*),
- $\Theta(n \log^k n)$, for some constant k (*quasi-linear*),
- $\Theta(n^2)$ (*quadratic*),
- $\Theta(n^3)$ (*cubic*),
- $\Theta(2^n)$ (*exponential*).

These are sorted in *increasing order* of growth rate.

Common growth rates

Commonly encountered growth rates in analysis of algorithms include the following:

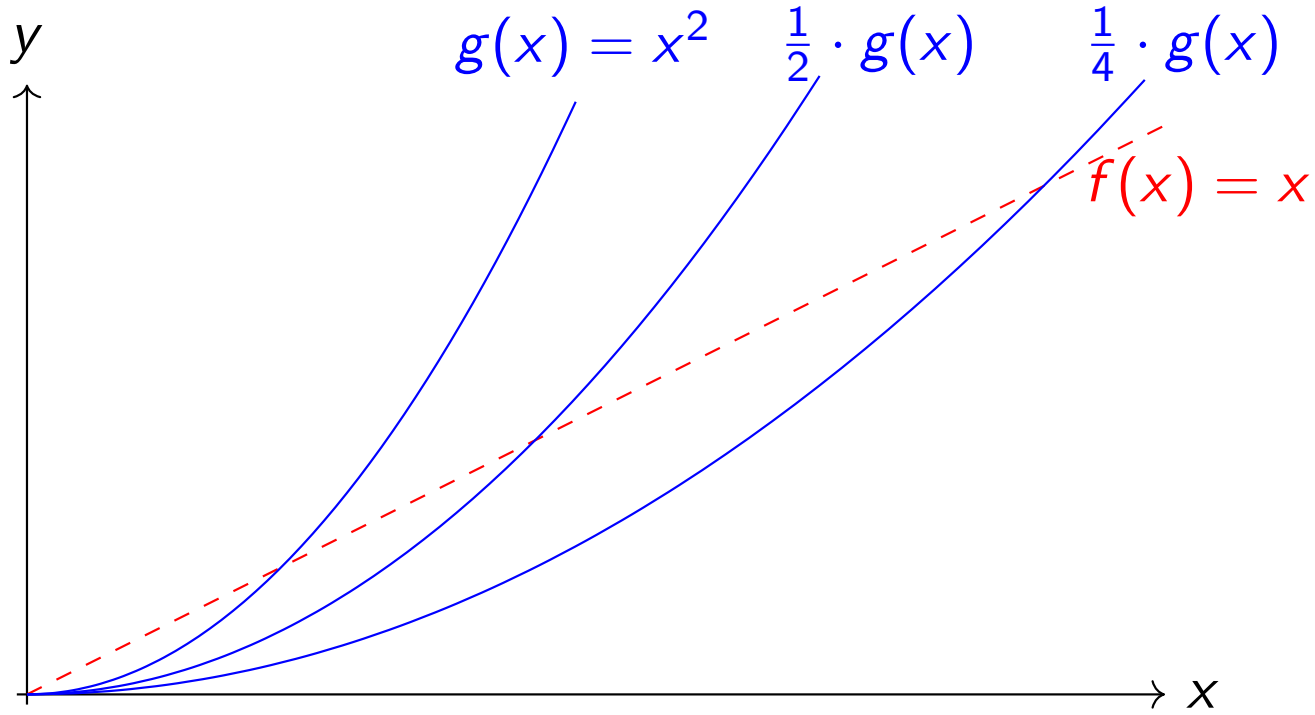
- $\Theta(1)$ (*constant*),
- $\Theta(\log n)$ (*logarithmic*),
- $\Theta(n)$ (*linear*),
- $\Theta(n \log n)$ (*linearithmic*),
- $\Theta(n \log^k n)$, for some constant k (*quasi-linear*),
- $\Theta(n^2)$ (*quadratic*),
- $\Theta(n^3)$ (*cubic*),
- $\Theta(2^n)$ (*exponential*).

These are sorted in *increasing order* of growth rate.

How do we define 'increasing order of growth rate'?

Strictly smaller asymptotic bounds

- We have $f(n) = n \in \Theta(n)$.
- How to express that $f(n)$ grows slower than n^2?



o-notation: $f(x) \in o(g(x))$ (f is *asymptotically strictly smaller* than g) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that $|f(x)| \leq c\,|g(x)|$ for all $x \geq n_0$.

Strictly smaller/larger asymptotic bounds

Example: Prove that $n \in o(n^2)$ from first principles.

Strictly smaller/larger asymptotic bounds

Example: Prove that $n \in o(n^2)$ from first principles.

- Main difference between o and O is the quantifier for c.
- n_0 will depend on c, so it is really a function $n_0(c)$.
- We also say 'the growth rate of f is *less than* the growth rate of g'.
- Rarely proved from first principles (instead use limit-rule $\rightsquigarrow$ later).

ω-notation: $f(x) \in \omega(g(x))$ (f is *asymptotically strictly larger* than g) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that $|f(x)| \geq c\,|g(x)|$ for all $x \geq n_0$.

- Symmetric, the growth rate of f is *more than* the growth rate of g.

Outline

The limit rule

Suppose that $f(x) > 0$ and $g(x) > 0$ for all $x \geq n_0$. Suppose that

$$L = \lim_{x \to \infty} \frac{f(x)}{g(x)} \quad \text{(in particular, the limit exists).}$$

Then

$$f(x) \in \begin{cases} o(g(x)) & \text{if } L = 0 \\ \Theta(g(x)) & \text{if } 0 < L < \infty \end{cases}$$

If the fraction tends to infinity then $f(x) \in \omega(g(x))$.

The required limit can often be computed using *l'Hôpital's rule*. Note that this result gives *sufficient* (but not necessary) conditions for the stated conclusion to hold.

Application 1: Logarithms vs. polynomials

Compare the growth rates of $f(n) = \log n$ and $g(n) = n$.

Now compare the growth rates of $f(n) = (\log n)^c$ and $g(n) = n^d$ (where $c > 0$ and $d > 0$ are arbitrary numbers).

Application 2: Polynomials

Let $f(n)$ be a polynomial of degree $d \geq 0$:

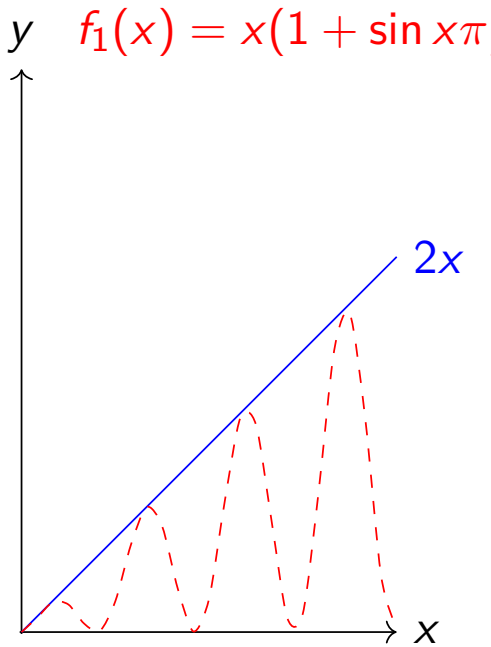
$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0$$

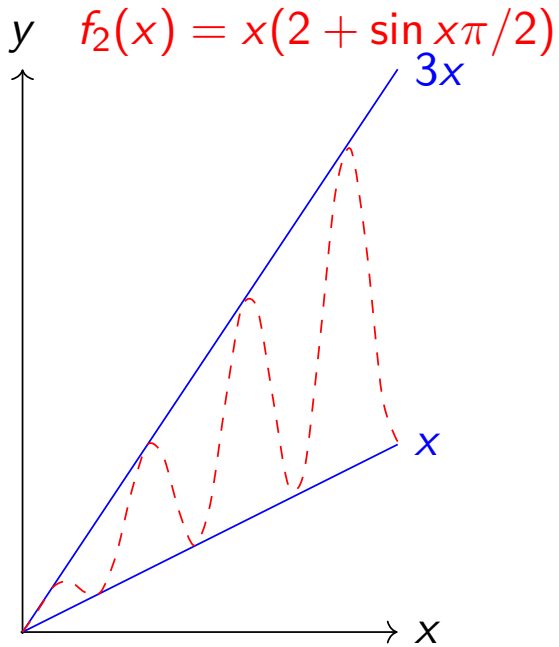
for some $c_d > 0$.

Then $f(n) \in \Theta(n^d)$:

Example: Oscillating functions

Consider two oscillating functions f_1, f_2 for which $\lim_{n\to\infty} \frac{f_i(x)}{x}$ does not exist. Are they in $\Theta(n)$?





So no limit $\rightsquigarrow$ must use other methods to prove asymptotic bounds.

Relationships between order notations

- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

Example: Fill the following table with TRUE or FALSE:

		Is $f(n) \in \ldots(g(n))$?			
$f(n)$	$g(n)$	o	O	Ω	ω
$\log n$	$\sqrt{n}$				

Asymptotic notation and arithmetic

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Avoid doing arithmetic with asymptotic notations.
  Do **not** write $O(n) + O(n) = O(n)$.
  (CS136 allowed you to be sloppy here. CS240 does not, mostly because it can go badly wrong with recursions.)
- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'
  (That's still a bit sloppy (why?), but less dangerous.)

Asymptotic notation and arithmetic

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Avoid doing arithmetic with asymptotic notations.
  Do **not** write $O(n) + O(n) = O(n)$.
  (CS136 allowed you to be sloppy here. CS240 does not, mostly because it can go badly wrong with recursions.)
- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'
  (That's still a bit sloppy (why?), but less dangerous.)
- There are some (very limited) exceptions:
  - $f(n) = n^2 + \Theta(n)$ means "$f(n)$ is n^2 plus a linear term"
    - ★ nicer to read than "$n^2 + n + \log n$"
    - ★ more precise about constants than "$\Theta(n^2)$"
  - But use this very sparingly (typically only for stating the final result)

Asymptotic notation and arithmetic

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Avoid doing arithmetic with asymptotic notations.
  Do **not** write $O(n) + O(n) = O(n)$.
  (CS136 allowed you to be sloppy here. CS240 does not, mostly because it can go badly wrong with recursions.)
- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'
  (That's still a bit sloppy (why?), but less dangerous.)
- There are some (very limited) exceptions:
  - $f(n) = n^2 + \Theta(n)$ means "$f(n)$ is n^2 plus a linear term"
    - ★ nicer to read than "$n^2 + n + \log n$"
    - ★ more precise about constants than "$\Theta(n^2)$"
  - But use this very sparingly (typically only for stating the final result)
  - Similarly $f(n) = n^2 + o(1)$ means "n^2 plus a vanishing term."

Outline

Back to: Algorithm analysis

4) Analyze the algorithm.

```
insertion-sort(A, n)
A: array of size n
1.  for (i ← 1; i < n; i++) do
2.      for (j ← i; j > 0 and A[j−1] > A[j]; j−−) do
3.          swap A[j] and A[j − 1]
```

- We want to bound the number of primitive operations
  $\rightsquigarrow$ count $\Theta(1)$ per line of code (unless it hides non-trivial work)
- Sum up work done over a loop.

Back to: Algorithm analysis

4) Analyze the algorithm.

```
insertion-sort(A, n)
A: array of size n
1.  for (i ← 1; i < n; i++) do
2.      for (j ← i; j > 0 and A[j−1] > A[j]; j−−) do
3.          swap A[j] and A[j − 1]
```

- We want to bound the number of primitive operations
  $\rightsquigarrow$ count $\Theta(1)$ per line of code (unless it hides non-trivial work)
- Sum up work done over a loop.
- The run-time depends on the specific instance (and on its size). E.g.
  - $A[i] \geq A[i-1] \rightsquigarrow$ inner loop takes $\Theta(1)$ time
  - $A[i] < A[0] \rightsquigarrow$ inner loop takes $\Theta(i)$ time

Let $T_{\mathcal{A}}(I)$ denote the running time of an algorithm $\mathcal{A}$ on instance I.

Study this value for the worst, best and 'typical' (average) instance I.

Complexity of algorithms

Worst-case (best-case) complexity of an algorithm: The *worst-case (best-case) running time* of an algorithm $\mathcal{A}$ is a function $T : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the *longest (shortest)* running time for any input instance of size n:

$$T_{\mathcal{A}}^{\text{worst}}(n) = \max_{I \in \mathcal{I}_n} \{ T_{\mathcal{A}}(I) \}$$

$$T_{\mathcal{A}}^{\text{best}}(n) = \min_{I \in \mathcal{I}_n} \{ T_{\mathcal{A}}(I) \}$$

To prove a lower bound on the worst-case run-time: Pick one especially bad example, and bound its run-time (using Ω-notation).

Average-case complexity of an algorithm: The average-case running time of an algorithm $\mathcal{A}$ is a function $T : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the *average* running time of $\mathcal{A}$ over all instances of size n:

$$T_{\mathcal{A}}^{avg}(n) = \sum_{I \in \mathcal{I}_n} T_{\mathcal{A}}(I) \cdot (\text{relative frequency of } I)$$

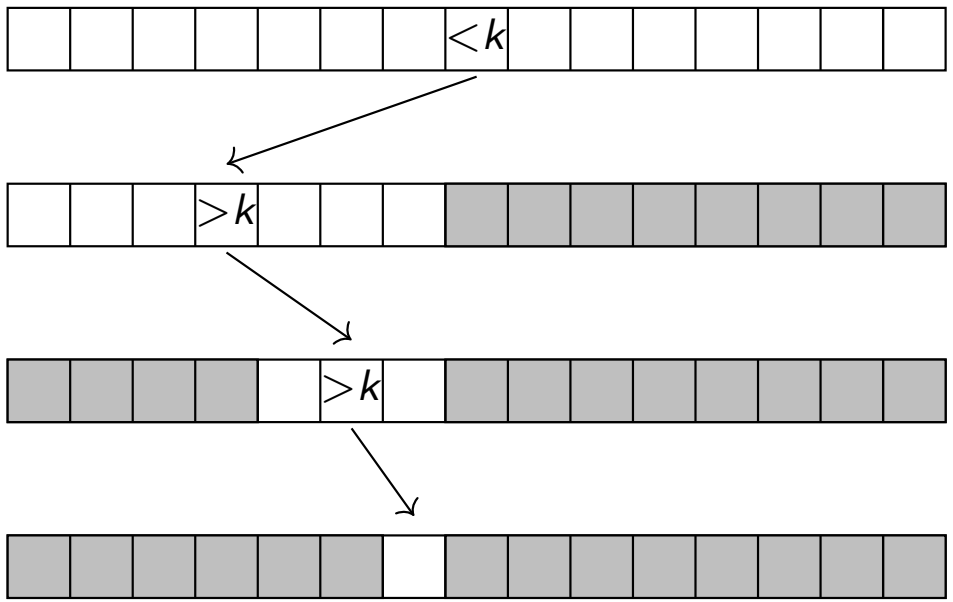
Analysis of recursive algorithms

Second example of 'solving a problem': recursive algorithm *binary search*.

Step 1: Describe the overall idea

Input: Sorted array A of n integers, one integer (**key**) k.
Goal: Test whether k is stored in A.

1. Look at the middle of A and compare to k.
2. This eliminate half of the array as possible places for k.
3. *Recursively* search in the remaining half until the array is very small.



Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

binary-search-recursive(A, n, k)
A: array of size n
1. **if** $(n \leq 0)$ **then return** "not found"
2. $m = \lfloor n/2 \rfloor$
3. **if** ($A[m]$ equals k) **then return** "found at $A[m]$"
4. **else if** $(A[m] < k)$ **then** *binary-search-recursive*$(A[0..m-1], m, k)$
5. **else** *binary-search-recursive*$(A[m+1..n-1], n-m+1, k)$

Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

binary-search-recursive(A, n, k)
A: array of size n
1. **if** $(n \leq 0)$ **then return** "not found"
2. $m = \lfloor n/2 \rfloor$
3. **if** ($A[m]$ equals k) **then return** "found at $A[m]$"
4. **else if** $(A[m] < k)$ **then** *binary-search-recursive*$(A[0..m-1], m, k)$
5. **else** *binary-search-recursive*$(A[m+1..n-1], n-m+1, k)$

Two tricks to reduce constant in the run-time and auxiliary space:

- Do not pass array A by value, instead indicate the sub-array $A[\ell..r]$ by keeping ℓ, r as parameters
- Even better: avoid recursion altogether, use a while-loop instead.

But for the analysis, the recursive version is sometimes easier to use.

Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

binary-search(A, n, k) `// the CS136 version`

A: Sorted array of size n, k: key

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** ($A[m]$ equals k) **then return** "found at $A[m]$"
5. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
6. $\quad$ **else** $r \leftarrow m-1$
7. **return** "not found, but would be between $A[\ell-1]$ and $A[\ell]$"

Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

binary-search(A, n, k) `// the CS136 version`

A: Sorted array of size n, k: key

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** ($A[m]$ equals k) **then return** "found at $A[m]$"
5. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
6. $\quad$ **else** $r \leftarrow m-1$
7. **return** "not found, but would be between $A[\ell-1]$ and $A[\ell]$"

But we can do **even** better:

- Humans automatically do 3-way comparisons ($<$, $=$, $>$)
- Computers do 2-way comparisons ($<$, $\geq$ or $=$, $\neq$)
- Tailor the algorithm to use 2-way comparisons.

Improving binary search

Step 2: Give pseudo-code or detailed description.

binary-search-optimized(A, n, k)
A: Sorted array of size n, k: key

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell < r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
5. $\quad$ **else** $r \leftarrow m$ `// this is different!`
6. **if** $(k = A[\ell])$ **then return** "found at $A[\ell]$"
7. **return** "not found"

Improving binary search

Step 2: Give pseudo-code or detailed description.

binary-search-optimized(A, n, k)
A: Sorted array of size n, k: key

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell < r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
5. $\quad$ **else** $r \leftarrow m$ `// this is different!`
6. **if** $(k = A[\ell])$ **then return** "found at $A[\ell]$"
7. **return** "not found"

Message: The same idea can be realized in many different ways.

- Usually (but not always) this does not affect asymptotic bounds.
- But it makes a big difference in an actual implementation.

Explaining the solution of a problem

Step 3: Argue correctness (here for *binary-search-optimized*):

- State loop-invariant: "k can only be in $A[\ell..r]$"
  - If not obvious: argue that it continues to hold
  - If not obvious: argue that it implies correct answer

Explaining the solution of a problem

Step 3: Argue correctness (here for *binary-search-optimized*):

- State loop-invariant: "k can only be in $A[\ell..r]$"
  - If not obvious: argue that it continues to hold
  - If not obvious: argue that it implies correct answer
- Important (especially for recursions): Why do we terminate?
  - Need to argue: size of problem becomes smaller.

Explaining the solution of a problem

Step 3: Argue correctness (here for *binary-search-optimized*):

- State loop-invariant: "k can only be in $A[\ell..r]$"
  - If not obvious: argue that it continues to hold
  - If not obvious: argue that it implies correct answer
- Important (especially for recursions): Why do we terminate?
  - Need to argue: size of problem becomes smaller.
  - Here: size $= r - \ell + 1 =$ number of keys that we search in.
  - Actually show: sub-array has size at most $\lceil (r - \ell + 1)/2 \rceil$.
  - And this is smaller than $r - \ell + 1$ if $\ell < r$.

Explaining the solution of a problem

Step 3: Argue correctness (here for *binary-search-optimized*):

- State loop-invariant: "k can only be in $A[\ell..r]$"
  - If not obvious: argue that it continues to hold
  - If not obvious: argue that it implies correct answer
- Important (especially for recursions): Why do we terminate?
  - Need to argue: size of problem becomes smaller.
  - Here: size $= r - \ell + 1 =$ number of keys that we search in.
  - Actually show: sub-array has size at most $\lceil (r - \ell + 1)/2 \rceil$.
  - And this is smaller than $r - \ell + 1$ if $\ell < r$.
- Step 3 is often interleaved with Step 2.
  - Which of the following should be m? $\lfloor \frac{r+\ell}{2} \rfloor, \lceil \frac{r+\ell}{2} \rceil, \lfloor \frac{r+\ell+1}{2} \rfloor, \ldots$?
  - Do the proof of correctness and see which one works.

Explaining the solution of a problem

Step 4: Analyze the run-time and space.

- First analyze work done outside recursions.
  - If applicable, analyze subroutines separately.
- If there are recursions: how big are the subproblems? The run-time then becomes a recursive function.

Let $T(n)$ denote the worst-case time on an array of length n.

- $\Theta(1)$ time to compute m, compare $A[m]$ to k
- Recurse in array of size n' where $\lfloor \frac{n-1}{2} \rfloor \leq n' \leq \lceil \frac{n-1}{2} \rceil$

Explaining the solution of a problem

Step 4: Analyze the run-time and space.

- First analyze work done outside recursions.
  - If applicable, analyze subroutines separately.
- If there are recursions: how big are the subproblems? The run-time then becomes a recursive function.

Let $T(n)$ denote the worst-case time on an array of length n.

- $\Theta(1)$ time to compute m, compare $A[m]$ to k
- Recurse in array of size n' where $\lfloor \frac{n-1}{2} \rfloor \leq n' \leq \lceil \frac{n-1}{2} \rceil$

The **recurrence relation** is as follows (constant factor c replaces Θ):

$$T(n) \leq \begin{cases} \max\left\{ T\left(\lceil \frac{n-1}{2} \rceil\right), T\left(\lfloor \frac{n-1}{2} \rfloor\right) \right\} + c & \text{if } n \geq 1 \\ c & \text{if } n = 0. \end{cases}$$

Explaining the solution of a problem

Step 4: Analyze the run-time and space.

- When $n = 2^p - 1$ for some integer p, then the exact recurrence can easily be solved by various methods.
  - E.g. prove by induction on p that $T(2^p - 1) \leq c(p+1) = c\log(2n+2)$.

Explaining the solution of a problem

Step 4: Analyze the run-time and space.

- When $n = 2^p - 1$ for some integer p, then the exact recurrence can easily be solved by various methods.
  - E.g. prove by induction on p that $T(2^p - 1) \leq c(p+1) = c\log(2n+2)$.
- For many functions, it is "good enough" to prove inequalities when n is "divisible as needed"
  - E.g.: If $f(n) \leq n^d$ for all powers of 2, and $f(n)$ is monotone then $f(n) \in O(n^d)$ (exercise)
- For this reason, we allow ourselves to use a **sloppy recurrence** that has floors, ceilings, small additive terms removed:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c & \text{if } n \geq 1 \\ c & \text{if } n = 0. \end{cases}$$

- Then prove (or look up) what this recurrence resolves to.

Some recurrence relations

Recursion	resolves to	example
$T(n) \leq T(n/2) + O(1)$	$T(n) \in O(\log n)$	binary-search
$T(n) \leq 2T(n/2) + O(n)$	$T(n) \in O(n \log n)$	merge-sort
$T(n) \leq 2T(n/2) + O(\log n)$	$T(n) \in O(n)$	heapify (*)
$T(n) \leq cT(n-1) + O(1)$ for some $c < 1$	$T(n) \in O(1)$	avg-case analysis (*)
$T(n) \leq 2T(n/4) + O(1)$	$T(n) \in O(\sqrt{n})$	range-search (*)
$T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$	$T(n) \in O(\sqrt{n})$	interpol. search (*)
$T(n) \leq T(\sqrt{n}) + O(1)$	$T(n) \in O(\log \log n)$	interpol. search (*)

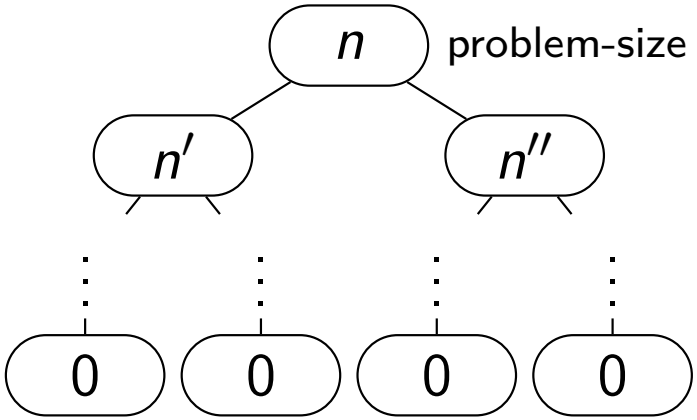
- Once you know the result, it is (usually) easy to prove by induction.
- These bounds are tight if the upper bounds are tight.
- Many more recursions, and some methods to find the result, in CS341.

(*) These may or may not get used later in the course.

Explaining the solution of a problem

Step 4: Analyze the run-time and space.

For time and space analysis, the **recursion tree** is often useful:



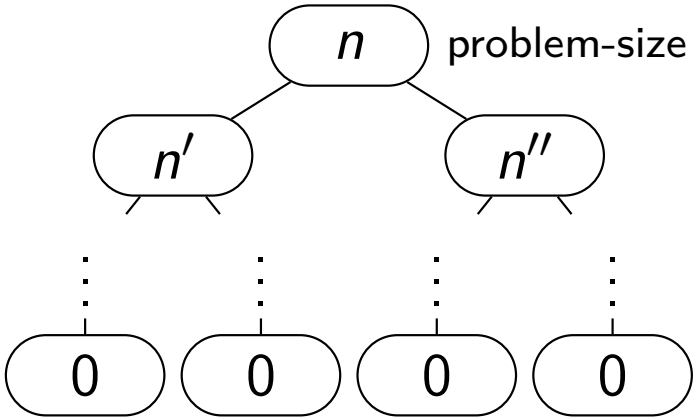
- Node = subproblem to solve.
  Child = recursive call from parent.
- For binary search, we recurse in only one subproblem

$\Rightarrow$ recursion tree is really a path.

Explaining the solution of a problem

Step 4: Analyze the run-time and space.

For time and space analysis, the **recursion tree** is often useful:



- Node = subproblem to solve.
  Child = recursive call from parent.
- For binary search, we recurse in only one subproblem

$\Rightarrow$ recursion tree is really a path.

- Recall: Each nested recursive call adds to the **recursion stack**.
- So the auxiliary space is Ω(height of the recursion tree)
- For *binary-search-recursive*:
  - Subproblem has size $n/2 + \Theta(1) \Rightarrow$ height is $\log n + \Theta(1)$.

  $\Rightarrow$ Use $\Theta(\log n)$ auxiliary space (for recursive version).
  - This also proves that the run-time is in $\Theta(\log n)$
    (We spend $\Theta(1)$ time on each level of the tree.)