Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Asymptotic notation, mergesort, recurrences

Outline

Review of the last lecture

- Introduced the problem of **sorting**.
- Analyzed `insertion-sort`.
  - Worst-case running time: $T(n) = \frac{3n^2}{2} + \frac{7n}{2} - 4$
  - Space: in-place algorithm
- Worst-case running time analysis: a reasonable measure of algorithmic efficiency.
- Defined polynomial-time algorithms as "efficient".
- Argued that detailed characterizations of running times are not convenient for understanding scalability of algorithms.

Running time in terms of # primitive steps

We need a coarser classification of running times of algorithms; exact characterizations

- are **too detailed**;
- do not reveal similarities between running times in an immediate way as n grows large;
- are often **meaningless**: high-level language steps will **expand** by a constant factor that depends on the hardware.

Today

1. Asymptotic notation

Can be tight or not tight

tight if the 2 functions you're comparing are growing @ the same rate.

not tight - one of the 2 functions is growing strictly faster than the other.

2. The divide & conquer principle; application: mergesort

3. Solving recurrences and running time of `mergesort`

Aymptotic analysis

A framework that will allow us to compare the rate of growth of different running times as the input size n grows.

- We will express the running time as a function of the number of primitive steps; the latter is a function of the input size n.
- To compare functions expressing running times, we will ignore their low-order terms and focus solely on the highest-order term.

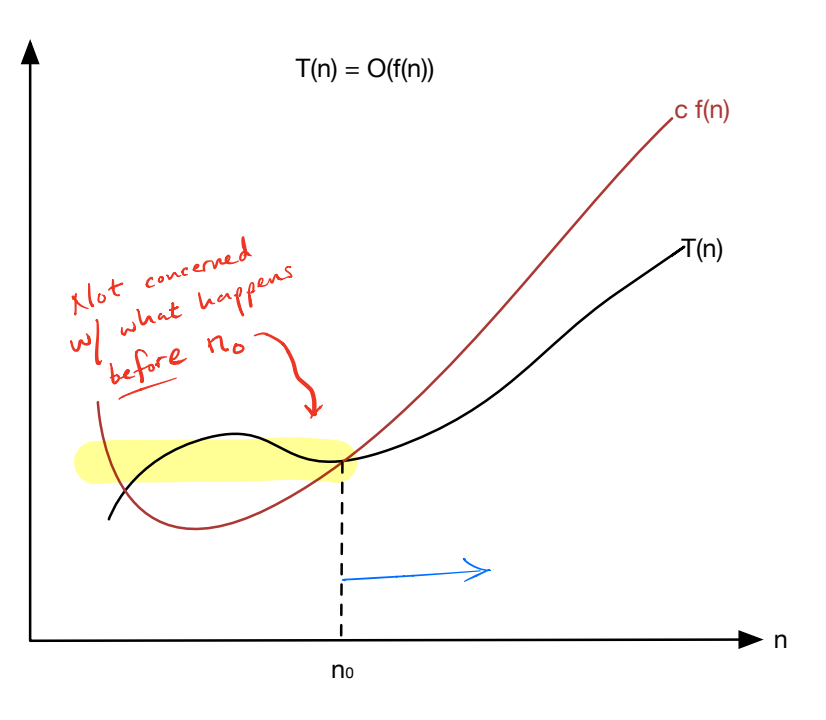
Asymptotic upper bounds: Big-O notation

Definition 1 (O).

We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

$T(n)$ grows at most as fast as $f(n)$

$T(n) = O(f(n))$ if $\exists$ some $c > 0$ and $n_0 \geq 0$ s.t. $\forall n \geq n_0$ we have $T(n) \leq c \cdot f(n)$



After n_0, $T(n)$ is always upperbounded by a constant multiple of $f(n)$.

Asymptotic upper bounds: Big-O notation

Definition 1 (O).

We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

Examples: Show that $T(n) = O(f(n))$ when

(1) ▸ $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.

(2) ▸ $T(n) = an^2 + b$ and $f(n) = n^3$.

(1) $an^2 + b \leq c \cdot n^2$ for all $n \geq n_0$

for your choice of c and n_0

$c = a + b$, $n_0 = 1$ There are many pairs (c, n_0) that satisfies this.

(2) $an^2 + b \leq c \cdot n^3$ for all $n \geq n_0$

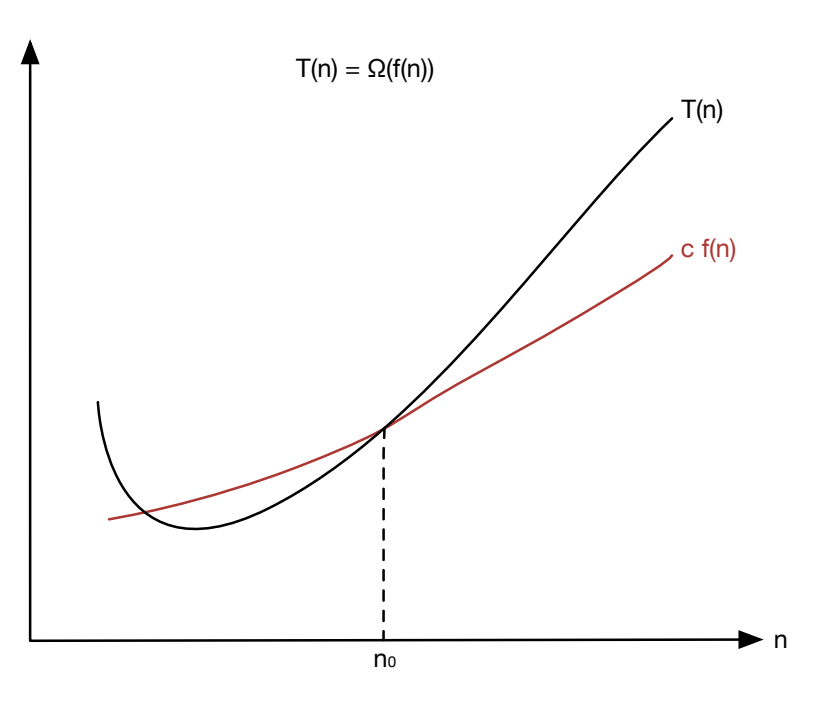
$c = a + b$, $n_0 = 1$ works here as well.

Asymptotic lower bounds: Big-Ω notation

Definition 2 (Ω).

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

T(n) grows at least as fast as f(n)



After n_0, T(n) is always lower-bounded by a constant multiple of f(n)

Asymptotic lower bounds: Big-Ω notation

Definition 2 (Ω).

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

Examples: Show that $T(n) = \Omega(f(n))$ when

(1) ▸ $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.

(2) ▸ $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n$.

(1) $an^2 + b \geq c \cdot n^2$ for all $n \geq n_0$

(c: ↑ a; n_0: ↑ 1)

$an^2 + b$ and n^2 grow @ the same rate, so of course you can lowerbound $an^2 + b$ by some constant multiple of n^2.

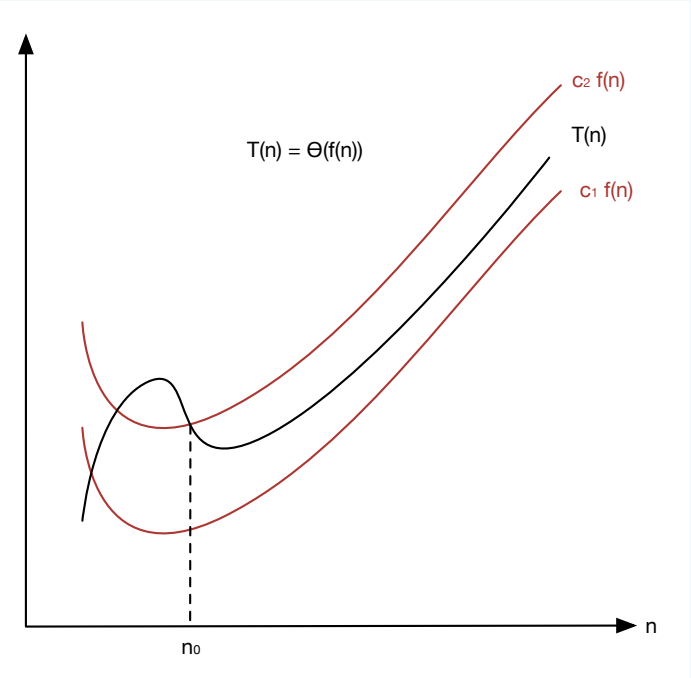
(2) $an^2 + b \geq c \cdot n$, use $c = a$ and $n_0 = 1$

Asymptotic tight bounds: Θ notation

Definition 3 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$



Asymptotic tight bounds: Θ notation

Definition 3 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$

Equivalent definition

$T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ *and* $T(n) = \Omega(f(n))$

T(n) grows at most as fast as f(n) and at least as fast as f(n) $\Longrightarrow$ T(n) and f(n) grow at the same rate.

Θ : a tight bound means that the 2 functions grow at the same rate.

Asymptotic tight bounds: Θ notation

Definition 3 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$

Equivalent definition

$T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ *and* $T(n) = \Omega(f(n))$

$c_1 n \log n \leq n \log n + n \leq c_2 n \log n$

$c_1 = 1$ $n_0 = 2$
$c_2 = 2$

Notational convention: $\log n$ stands for $\log_2 n$

Examples: Show that $T(n) = \Theta(f(n))$ when

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$
- $T(n) = n \log n + n$ and $f(n) = n \log n$

Have shown that
$T(n) = O(f(n))$
$T(n) = \Omega(f(n))$

The $n\log_2 n$ term grows much faster than the n term which becomes negligible compared to $n\log_2 n$ as the input size increases.

$$c_1 \cdot n\log n \le n\cdot\log n + n \le c_2 \cdot n\log n$$

$\forall n \ge n_0$

Pick $c_1 = 1$, $c_2 = 2$, $n_0 = 2$:

$$n\log n \le n\log n + n \le 2n\log n$$
$$= n\log n + n\log n$$

$\forall n \ge 2$

Asymptotic upper bounds that are **not** tight: little-o

Definition 4 (o).

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

↳ No matter how tiny you pick c, $T(n)$ will always be less than $c \cdot f(n)$ for a large enough input size.

$\Rightarrow$ $f(n)$ grows MUCH faster than $T(n)$

Asymptotic upper bounds that are **not** tight: little-o

Definition 4 (o).

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

- Intuitively, $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- Proof by showing that $\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Asymptotic upper bounds that are **not** tight: little-o

When we look for improvements in our algorithms, we will be looking for running times that compare in this fashion. The new running time will be little-o of the old running time. Because this is a meaningful improvement.

Definition 4 (o).

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

- Intuitively, $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- Proof by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Examples: Show that $T(n) = o(f(n))$ when

(1) ▸ $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^3$.

(2) ▸ $T(n) = n \log n$ and $f(n) = n^2$. →

(1) $\lim_{n\to\infty} \frac{an^2+b}{n^3} = 0$

(2) $\lim_{n\to\infty} \frac{n\log n}{n^2} = \lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{1}{n} = 0$

Asymptotic lower bounds that are **not** tight: little-ω

Definition 5 (ω).

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

* $T(n)$ grows much faster than $f(n)$ no matter the value of c.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 5 (ω).

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

- Intuitively $T(n)$ becomes arbitrarily large relative to $f(n)$, as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$, if the limit exists. Then $f(n) = o(T(n))$.
- Equivalently, we can show that $\lim_{n\to\infty} \frac{f(n)}{T(n)} = 0$

If the limit of $\frac{T(n)}{f(n)}$ or $\frac{f(n)}{T(n)}$ is some constant c, then $T(n) = \Theta(f(n))$ and $f(n) = \Theta(T(n))$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 5 (ω).

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

- Intuitively $T(n)$ becomes arbitrarily large relative to $f(n)$, as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$, if the limit exists. Then $f(n) = o(T(n))$.

Examples: Show that $T(n) = \omega(f(n))$ when

- $T(n) = n^2$ and $f(n) = n \log n$. → Previously shown that $n \log n = o(n^2)$ $\Rightarrow n^2 = \omega(n \log n)$
- $T(n) = 2^n$ and $f(n) = n^5$.

$\lim_{n\to\infty} \frac{2^n}{n^5} \xrightarrow{L'H} \lim_{n\to\infty} \frac{2^n \cdot c}{5!} = \infty$ $\Rightarrow 2^n = \omega(n^5)$

Basic rules for omitting low order terms from functions

1. Ignore **multiplicative** factors: e.g., $10n^3$ becomes n^3
2. n^a dominates n^b if $a > b$: e.g., n^2 dominates n
3. Exponentials dominate polynomials: e.g., 2^n dominates n^4
4. Polynomials dominate logarithms: e.g., n dominates $\log^3 n$

$\Rightarrow$ For large enough n,

$$\log n < n < n \log n < n^2 < 2^n < 3^n < n^n$$

$n! = 1 \cdot 2 \cdot 3 \cdots n$

$3^3 = 27$

$3! = 6$

$3^3 = 27$

$3! = 6$

Properties of asymptotic growth rates

$2n = O(n)$
$5n = O(n)$
$7n = O(n)$
} $2n + 5n = O(n)$ and $2n + 5n + 7n = O(n)$

1. **Transitivity**
   - 1.1 If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
   - 1.2 If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
   - 1.3 If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.
2. **Sums** of up to a constant number of functions
   - 2.1 If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
   - 2.2 Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k, h$ be functions such that for all i, $f_i = O(h)$. Then $f_1 + f_2 + \ldots + f_k = O(h)$.

Must be a constant number of functions.

3. **Transpose symmetry**
   - $f = O(g)$ if and only if $g = \Omega(f)$.
   - $f = o(g)$ if and only if $g = \omega(f)$.

$c \cdot O(f) = O(f)$

$O(f) \cdot O(g) = O(fg)$

Indeed, $O(f) = c \cdot f$
$O(g) = c'g$
ef $c'g = c''fg = O(fg)$

Today

1. Asymptotic notation
2. The divide & conquer principle; application: mergesort
3. Solving recurrences and running time of `mergesort`

The divide & conquer principle

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems to get the solution to the overall problem.

Divide & Conquer applied to sorting

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  Divide the input array into two lists of equal size.
- **Conquer** the subproblems by solving them recursively.
  Sort each list recursively. (Stop when lists have size 2.)
- **Combine** the solutions to the subproblems into the solution for the original problem.
  Merge the two sorted lists and output the sorted array.

mergesort: pseudocode

1 n

```
mergesort (A, left, right)
    if right == left then return
    end if
    mid = left + ⌊(right − left)/2⌋
    mergesort (A, left, mid)
    mergesort (A, mid + 1, right)
    merge (A, left, right, mid)
```

← if size of array is 1

All the work occurs in merge(·)

↑ Extra Space req'd: n

↑ merge (A, left, mid, right) ?

Extra Space req'd by mergesort : n (worse than insertion sort)

(Because you can reuse space.)

Remarks

- `mergesort` is a recursive procedure *(why?)*
  Because it calls itself. mergesort sorts by calling itself. It doesn't call any other sorting algorithm.
- Initial call: `mergesort`$(A, 1, n)$
- Subroutine `merge` merges two sorted lists of sizes $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ into one sorted list of size n. *How can we accomplish this?*

Prove correctness of mergesort by induction.

Proof of correctness of mergesort (By induction)

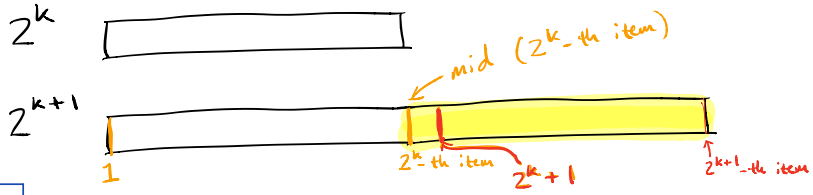
Pf: Spz $n = 2^k$ (In HW have to prove for ANY input size)

Base Case: $k = 0 \Rightarrow n = 1$

Is mergesort correct on $n=1$? Yes, it does nothing.

Ind. Hyp.: Mergesort sorts correctly for some input size $n = 2^k \geq 1$

Step: Show mergesort sorts correctly for input size 2^{k+1}



$(2 \cdot 2^k = 2^{k+1})$

Run mergesort on the first half. The output would be the sorted half since mergeSort is correct on input size 2^k.

Then run mergesort on the second half, which is also of size 2^k since $2^{k+1} = 2 \cdot 2^k = 2^k + 2^k$

Run mergesort on list/array $(2^k+1, 2^{k+1})$, and again mergesort will have sorted correctly since the input size of second half is 2^k.

Next, merge the two sorted halves into a sorted output, and you know that this will be done correctly because merge is correct (which we have to prove on our own - that merge is correct).

$mid = 2^k$

$$mid = 1 + \lfloor (2^{k+1}-1)/2 \rfloor = 1 + \lfloor 2^k - \tfrac{1}{2} \rfloor = 1 + 2^k - 1 = 2^k$$

$\Rightarrow mid = 2^k$

```
Mergesort (A, left, right)
    if right == left then return
    end if
    mid = left + ⌊(right − left)/2⌋
    Mergesort (A, left, mid)
    Mergesort (A, mid + 1, right)
    Merge (A, left, right, mid)
```

$mid = 2^k$

mergesort $(A, 1, 2^k)$

mergesort $(A, 2^k + 1, 2^{k+1})$

Merge $(A, 1, 2^{k+1}, 2^k)$

The output from each mergesort( ) call would be the sorted arrays, since by the Induction Hypothesis, mergesort correctly sorts arrays up to size 2^k.

Running Time of mergesort

$n \log n$: why?

* Because merge takes $O(n)$ time and you call it $\log n$ times.

Let $T(n)$ be the running time of mergesort on input size n.

$$T(n) = c + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c''n$$

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

↑ absorbed all constants

Still need to determine asymptotic solution. Have to solve the recurrence expression (*)

since we are only interested in asymptotic solution.

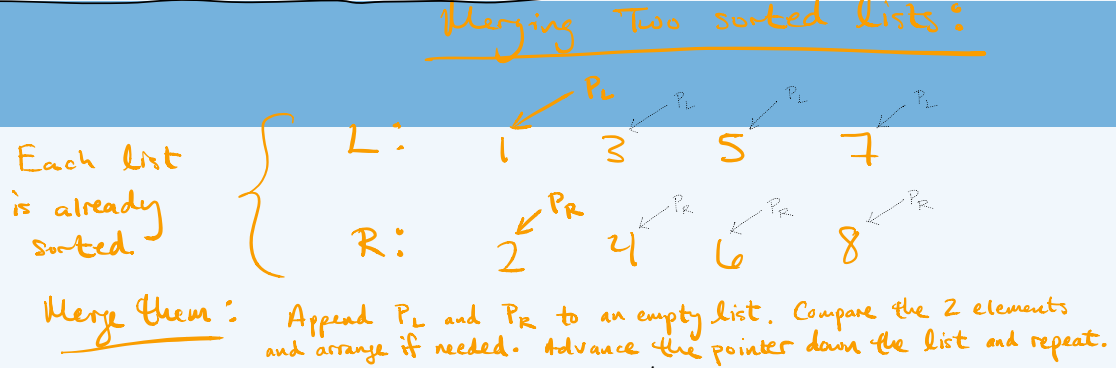
Because we are running mergesort on half the input size.

```
mergesort (A, left, right)
c      → if right == left then return
         end if
c      → mid = left + ⌊(right − left)/2⌋
T(n/2) → mergesort (A, left, mid)
T(n/2) → mergesort (A, mid + 1, right)
O(n)   → merge (A, left, right, mid)
```

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

$$T(1) = 2T\left(\frac{1}{2}\right) + c = c$$

merge: intuition



Intuition: To merge two sorted lists of size $n/2$ repeatedly

- compare the two items in the front of the two lists;
- extract the smaller item and append it to the output;
- update the front of the list from which the item was extracted.

Example: $n = 8, L = \{1, 3, 5, 7\}, R = \{2, 6, 8, 10\}$

merge: pseudocode

Say 1 n n/2

This implements what is described above. This pseudocode is acceptable.

merge $(A, left, right, mid)$

→ $L = A[left, mid]$ (1, n/2)

→ $R = A[mid + 1, right]$ (n/2 + 1, n)

Copy each element of A in these lists

Extra Space Requirement: n

Then what are the space requirements of merge on input size n? n

n: it allocates n + constant number of memory locations, but it's O(n).

Here we need n extra memory locations because we are copying the entire input to 2 lists.

Maintain two pointers p_L, p_R, initialized to point to the first elements of L, R, respectively

while both lists are nonempty **do**

Let x, y be the elements pointed to by p_L, p_R

Compare x, y and append the smaller to the output

Advance the pointer in the list with the smaller of x, y

end while

Append the remainder of the non-empty list to the output.

Remark: the output is stored directly in $A[left, right]$, thus the subarray $A[left, right]$ is sorted after **merge**$(A, left, right, mid)$.

How to prove correctness?

A: Induction on the size of the 2 lists

[Will typically prove correctness by induction.]

merge: pseudocode

TIME Requirements:

merge $(A, left, right, mid)$

→ $L = A[left, mid]$ ← Time required? You are copying over $n/2$ memory locations into the array L. This requires $O(n)$ time.

→ $R = A[mid+1, right]$ ← Also requires $O(n)$ time.

Maintain two pointers p_L, p_R, initialized to point to the first → $O(1)$ elements of L, R, respectively

Initialization of pointers requires $O(1)$ time.

while both lists are nonempty **do** ← Initializing the pointers: $O(1)$

$O(1)$ → Let x, y be the elements pointed to by p_L, p_R

$O(1)$ → Compare x, y and append the smaller to the output

$O(1)$ → Advance the pointer in the list with the smaller of x, y

end while $n \cdot O(1) = O(n)$

$O(n)$ → Append the remainder of the non-empty list to the output.

- How much time spent in the while-loop?
  * You can enter the while-loop at most n times.
  * Individual steps each require constant time
  ⇒ Total time in while-loop: $n \cdot O(1) = O(n)$

$O(n)$: Because in the worst case, your lists could be 1 2 3 4 5 6 7 8. The top list would append first. Remainder: $n/2$

Remark: the output is stored directly in $A[left, right]$, thus the subarray $A[left, right]$ is sorted after merge$(A, left, right, mid)$.

Total Time in Algorithm: $O(n) + O(n) + O(1) + O(n) + O(n) = O(n)$

merge: pseudocode

Space Requirement : $O(n)$ – It reuses space, but is not an in-place algorithm.

`merge` $(A, left, right, mid)$

$L = A[left, mid]$
$R = A[mid + 1, right]$
Maintain two pointers p_L, p_R, initialized to point to the first elements of L, R, respectively
while both lists are nonempty **do**
    Let x, y be the elements pointed to by p_L, p_R
    Compare x, y and append the smaller to the output
    Advance the pointer in the list with the smaller of x, y
end while
Append the remainder of the non-empty list to the output.

Remark: the output is stored directly in $A[left, right]$, thus the subarray $A[left, right]$ is sorted after `merge`$(A, left, right, mid)$.

`merge`: optional exercises

Optional exercise 1: write detailed pseudocode or actual code for `merge`

Optional exercise 2: write a recursive `merge`

Analysis of `merge`

1. **Correctness**

2. **Running time**

3. **Space**

Analysis of `merge`: correctness

1. **Correctness:** by induction on the size of the two lists (recommended exercise)
2. **Running time**

3. **Space**

`merge`: pseudocode

`merge` $(A, left, right, mid)$

$L = A[left, mid]$ →**not** a primitive computational step!
$R = A[mid + 1, right]$ →**not** a primitive computational step!
Maintain two pointers p_L, p_R initialized to point to the first elements of L, R, respectively
while both lists are nonempty **do**
    Let x, y be the elements pointed to by p_L, p_R
    Compare x, y and append the smaller to the output
    Advance the pointer in the list with the smaller of x, y
end while
Append the remainder of the non-empty list to the output.

Remark: the output is stored directly in $A[left, right]$, thus the subarray $A[left, right]$ is sorted after `merge`$(A, left, right, mid)$.

Analysis of `merge`: running time

1. **Correctness:** by induction on the size of the two lists (recommended exercise)
2. **Running time:**
   - Suppose L, R have $n/2$ elements each
   - *How many iterations before all elements from both lists have been appended to the output?*
   - *How much work within each iteration?*
3. **Space**

Analysis of `merge`: space

1. **Correctness:** by induction on the size of the two lists (recommended exercise)
2. **Running time:**
   - L, R have $n/2$ elements each
   - *How many iterations before all elements from both lists have been appended to the output?* At most $n - 1$.
   - *How much work within each iteration?* Constant.

   $\Rightarrow$ `merge` takes $O(n)$ time to merge L, R (*why?*).
3. **Space:** extra $\Theta(n)$ space to store L, R (the output of `merge` is stored directly in A).

Refreshing your memory on recursive algorithms

Exercise (recommended): run `mergesort` on input $1, 7, 4, 3, 5, 8, 6, 2$.

Analysis of `mergesort`

1. **Correctness**
2. **Running time**
3. **Space**

`mergesort`: correctness

For simplicity, assume $n = 2^k$ for integer $k \geq 0$.
We will use induction on k.

- **Base case:** For $k = 0$, the input consists of 1 item; `mergesort` returns the item.
- **Induction Hypothesis:** For $k \geq 0$, assume that `mergesort` correctly sorts any list of size 2^k.
- **Induction Step:** We will show that `mergesort` correctly sorts any list A of size 2^{k+1}.

  From the pseudocode of `mergesort`, we have:

  - Line 3: *mid* takes the value 2^k
  - Line 4: `mergesort`$(A, 1, 2^k)$ correctly sorts the leftmost half of the input, by the induction hypothesis.
  - Line 5: `mergesort`$(A, 2^k + 1, 2^{k+1})$ correctly sorts the rightmost half of the input, by the induction hypothesis.
  - Line 6: `merge` correctly merges its two sorted input lists into one sorted output of size $2^k + 2^k$.

  $\Rightarrow$ `mergesort` correctly sorts any input of size 2^{k+1}.

Running time of `mergesort`

Much more efficient than insertion sort.

Note: $n \log n = o(n^2)$ (little-o)

The running time of `mergesort` satisfies:

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$
$$T(1) = c$$

Therefore, this alg. is MUCH more efficient than insertion sort.

For $n = 10^6$:
$n \log n$ requires $20 \cdot 10^6$
n^2 requires $10^6 \cdot 10^6$.

This structure is typical of recurrence relations

- an **inequality** or **equation** bounds $T(n)$ in terms of an expression involving $T(m)$ for $m < n$
- a base case generally says that $T(n)$ is constant for small constant n

Remarks

- We ignore floor and ceiling notations.
- A recurrence does **not** provide an asymptotic bound for $T(n)$: to this end, we must **solve** the recurrence.

We don't know just by looking that $T(n) = O(n \log n)$.

Need to solve the recurrence to determine this.

Today

1. Asymptotic notation
2. The divide & conquer principle; application: mergesort
3. Solving recurrences and running time of `mergesort`

Solving recurrences, method 1: recursion trees

The technique consists of three steps

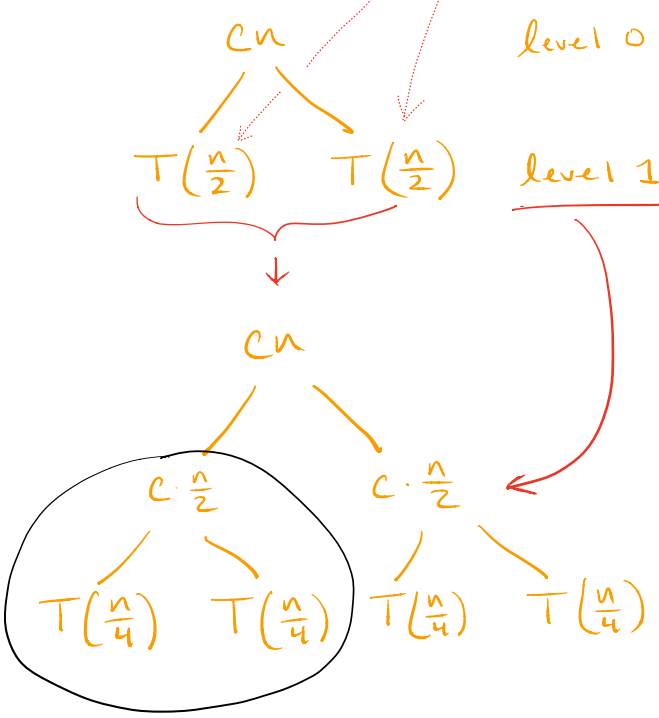
1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum the work spent over all levels of recursion

Example: give an asymptotic bound for the recurrence describing the running time of `mergesort`

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$
$$T(1) = c$$

$$T(n) = 2T(n/2) + cn,$$
$$T(1) = c$$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + c \cdot \frac{n}{2}$



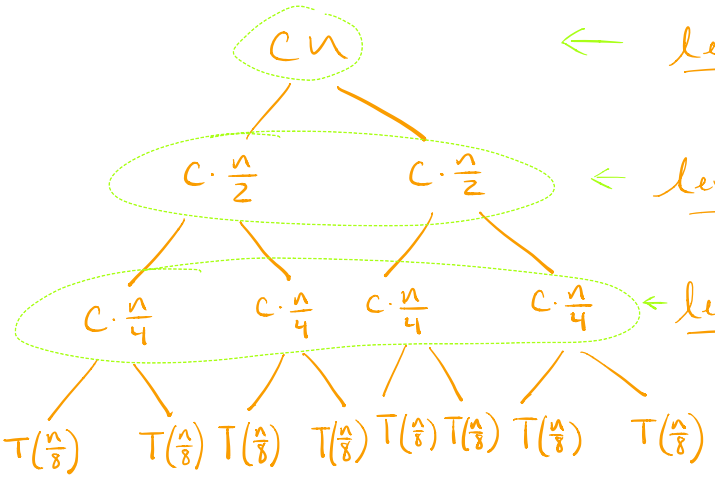
level 0 : 1 subproblem, size: n, total work: cn

level 1 : 2 subproblems, each of size: $\frac{n}{2}$, Total work: $c \cdot \frac{n}{2} + c \cdot \frac{n}{2} = cn$

→ Keep expanding in this way and you are eventually left with only expressions that are in terms of n.

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + c \cdot \frac{n}{4}$

Then,

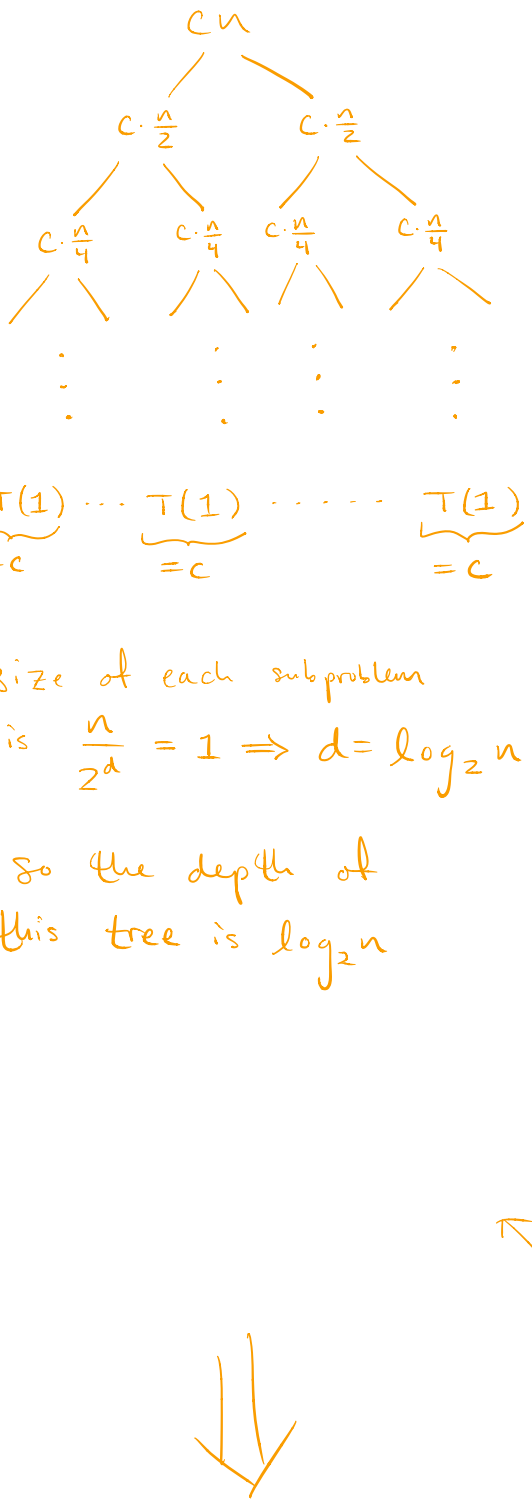


← level 0 : 1 subproblem, size: n, total work: cn

← level 1 : 2 subproblems, each: size n/2, total work: cn

← level 2 : 4 subproblems, each: size n/4, total work: cn

⋮



Size of each subproblem is $\frac{n}{2^d} = 1 \Rightarrow d = \log_2 n$

So the depth of this tree is $\log_2 n$

level i: 2^i subproblems

each: size $n/2^i$

total work: cn

$\left(= 2^i \cdot c \cdot \frac{n}{2^i} = cn\right)$

The tree will eventually bottom out at some level d. It bottoms out because at the bottom level, each subproblem will have size 1. At that point, mergesort will just terminate.

We reach level d when the size of the subproblem is 1.

---

level d: 2^d subproblems

each size: $n/2^d = 1$

$\Rightarrow n = 2^d$

$\Rightarrow \log_2 n = \log_2 2^d$

$\Leftrightarrow d = \log_2 n$

$T(n)$ is the sum of ALL the work spent on every level of the tree:

$$T(n) = \sum_{i=0}^{d} cn = (d+1) \cdot cn$$

$$= (\log_2 n + 1) \cdot cn$$

$$= \Theta(n \log n)$$

Therefore, the running time of mergesort is $\Theta(n \log n)$.

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

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

$= 4\left(2T(\frac{n}{8}) + c\frac{n}{4}\right) + 2cn$

$= 8T(\frac{n}{8}) + cn + 2cn = 8T(\frac{n}{8}) + 3cn$

$= T(n)$

$T(n) = 8\left(2T(\frac{n}{16}) + c\frac{n}{8}\right) + 3cn = 16T(\frac{n}{16}) + 4cn \ \ldots$

A general recurrence and its solution

The running times of many recursive algorithms can be expressed by the following recurrence

$$T(n) = aT(n/b) + cn^k, \text{for } a, c > 0,\ b > 1, k \geq 0$$

What is the recursion tree for this recurrence?

- a is the branching factor
- b is the factor by which the size of each subproblem shrinks

$\Rightarrow$ at level i, there are a^i subproblems, each of size n/b^i

$\Rightarrow$ each subproblem at level i requires $c(n/b^i)^k$ work

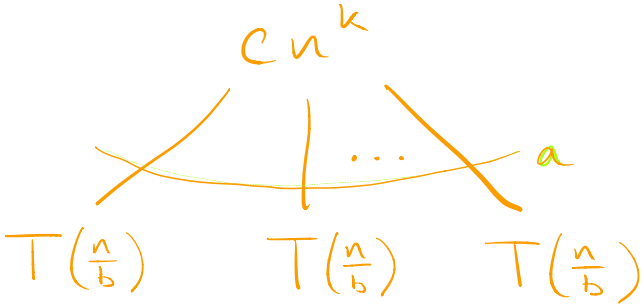
- the height of the tree is $\log_b n$ levels

$\Rightarrow$ Total work: $\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k,$$

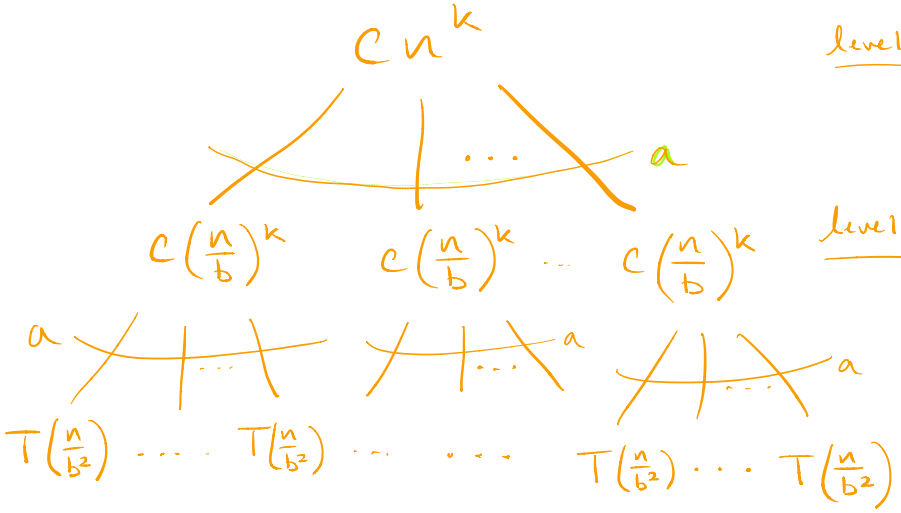
$T\left(\frac{n}{b}\right) = a\,T\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k$

$T\left(\frac{n}{b^2}\right) = a\,T\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^k$



level 0: 1 subproblem
size: n
total work: cn^k

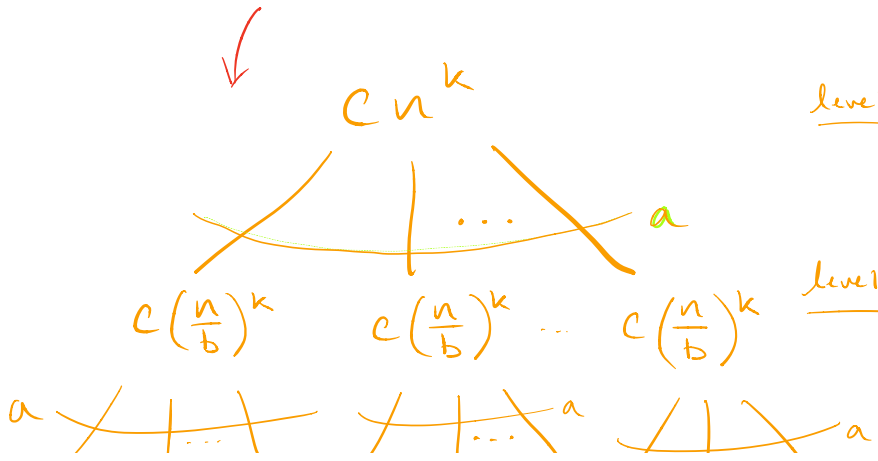
level 1: a subproblems



level 0: 1 subproblem
size: n
total work: cn^k

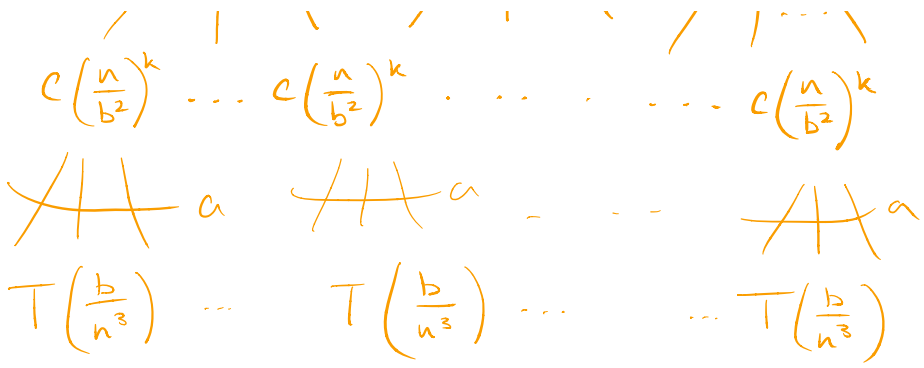
level 1: a subproblems
each: size $\frac{n}{b}$
total work: $a \cdot c\left(\frac{n}{b}\right)^k$

[Indeed, we spend $c\cdot\left(\frac{n}{b}\right)^k$ per subproblem, and there are 'a' of them.]



level 0: 1 subproblem
size: n
total work: cn^k

level 1: a subproblems
each: size $\frac{n}{b}$
total work: $a \cdot c\left(\frac{n}{b}\right)^k$



level 2: a^2 subproblems

each: size $\frac{n}{b^2}$

total work: $a^2 \cdot c\left(\frac{n}{b^2}\right)^k$

⋮

level i: a^i subproblems

each: size $\frac{n}{b^i}$

total work: $a^i \cdot c\left(\frac{n}{b^i}\right)^k$

Bottoms out at some level d, where each subproblem is of size

$$\frac{n}{b^d} = 1 \Rightarrow d = \log_b n$$

Tree bottoms out at some level d, where each subproblem is of size 1.

level d: a^d subproblems

each size $n/b^d = 1$

$\Rightarrow d = \log_b n$

total work: $T(n)$

$$T(n) = \sum_{i=0}^{d} a^i \cdot c \cdot \left(\frac{n}{b^i}\right)^k = cn^k \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$$

$$\Leftrightarrow \boxed{T(n) = cn^k \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i}$$

$i = 0$

- If $a = b^k \Rightarrow T(n) = (\log_b n + 1) \cdot cn^k$

$$= O(n^k \log_b n)$$

$$T(n) = cn^k \cdot \sum_{i=0}^{\log_b n} 1 = (\log_b n + 1) \cdot cn^k$$

We can replace $\log_b n$ with $\log_2 n$ in asymptotic notation (Check)

Don't need to keep $\log_b(\cdot)$ in base b notation (Why?)

$$= O(n^k \log n)$$

- If $a < b^k$ (In Homework) $\Rightarrow \dots \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i = O(1)$

$$\Rightarrow T(n) = O(n^k)$$

- If $a > b^k \Rightarrow \dots \sum \left(\frac{a}{b^k}\right)^i = O\left(\left(\frac{a}{b^k}\right) \log_b n\right)$

$$= O\left(\frac{a^{\log_b n}}{b^{k \log_b n}}\right)$$

$$= O\left(\frac{n^{\log_b a}}{n^k}\right)$$

$$\Rightarrow T(n) = O(n^{\log_b a})$$

Recall / Review:

$$\log_b n = \frac{\log_2 n}{\log_2 b}$$

and

$$\frac{\log_2 n}{\log_2 b} > \log_2 n$$

but only by a constant factor.

For $a > b^k$: You are summing terms, of a geometric progression, that are continuously increasing. Therefore, the last term of the (finite) sum $\sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$ is the largest.

Hence,

$$\sum \left(\frac{a}{b^k}\right)^i = \Theta\left(\left(\frac{a}{b^k}\right)^{\log_b n}\right) = \Theta\left(\frac{n^{\log_b a}}{n^k}\right)$$

$$\left(\frac{a}{b^k}\right)^{\log_b n} = \frac{a^{\log_b n}}{b^{\log_b n^k}} = \frac{n^{\log_b a}}{n^k}$$

and so $$cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i = cn^k \Theta\left(\frac{n^{\log_b a}}{n^k}\right)$$

$$\Rightarrow T(n) = O\left(n^{\log_b a}\right)$$

Solving recurrences, method 2: Master theorem

Theorem 6 (Master theorem).

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ *for some constants* $a > 0$, $b > 1$, $k \geq 0$, *then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \text{ if } a > b^k \\ O(n^k \log n) & , \text{ if } a = b^k \\ O(n^k) & , \text{ if } a < b^k \end{cases}$$

Example: running time of `mergesort`

- $T(n) = 2T(n/2) + cn$:
  $a = 2, b = 2, k = 1, b^k = 2 = a \Rightarrow T(n) = O(n \log n)$

Solving recurrences, method 3: the substitution method

The technique consists of two steps

1. Guess a bound
2. Use (strong) induction to prove that the guess is correct

(See your textbook for more details on this technique.)

Remark 1 (simple vs strong induction).

1. **Simple induction:** *the induction step at n requires that the inductive hypothesis holds at step $n-1$.*
2. **Strong induction** *is just a variant of simple induction where the induction step at n requires that the inductive hypothesis holds at all previous steps* $1, 2, \ldots, n-1$.

How would you solve...

HINT: Use recursion trees to solve.

1. $T(n) = 2T(n-1) + 1, T(1) = 2$

2. $T(n) = 2T^2(n-1), T(1) = 4$

3. $T(n) = T(2n/3) + T(n/3) + cn$

Ex: $T(n) = \underset{a}{4}\, T\left(\frac{n}{\underset{b}{4}}\right) + cn^{\overset{k}{}}$

$a = 4$
$b = 4$
$k = 1$

$b^k = 4 \Rightarrow a = b^k$

$\Rightarrow T(n) = O(n^k \log n)$

$= O(n \log n)$