



UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste – a.a. 2026/2027

Prof. Antonio Emanuele Cinà

`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences

University of Trieste

a.a. 2026/2027

Course material

Team code: **hro0fkt**

Moodle: **298SM - ALGORITHMIC DESIGN 2026**

Rooms and schedule

Monday: Room 3B, building H2bis, 4-6pm

Tuesday: Room 5C, building H2bis, 11am-1pm

Tuesday: Room 3A, H2bis, 5-7pm
(exception: 06/10/2026, Room 4 INF01, building D, 5-7pm)

Thursday: Room 5A, building H2bis, 1-3pm.

Chapter 3:

Recurrences

From recursive calls to running time

Chapter 1 introduced recursive algorithms; Chapter 2 supplied cost models and asymptotic notation.

We now translate recursive calls into a recurrence and solve it for a nonrecursive bound.

Iteration Expand calls and identify a general pattern.

Recursion trees Organize the work by depth.

Master Theorem Recognize common balanced cases.

Binary search connects the cost analysis to correctness arguments.

What a running-time recurrence records

Definition (Running-time recurrence)

A *running-time recurrence* expresses the cost of solving an instance of size n in terms of the costs of solving strictly smaller instances, together with the work performed outside the recursive calls and one or more base cases.

1. How many recursive calls are executed?
2. What is the input size of each call?
3. How much work occurs outside recursion?

Write $T(n)$ for the running time on an input of size n . Local work includes tests, index calculations, decomposition, and combination.

State the base-case costs as well.

Count the calls actually executed

Direct Fibonacci

Two calls, on different sizes; constant additional work:

$$T(n) = T(n-1) + T(n-2) + \Theta(1).$$

Binary search

One call on approximately half of the active array:

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1).$$

Merge sort

Two calls and linear merging work, for powers of two:

$$T(n) = 2T(n/2) + \Theta(n).$$

The divide-and-conquer form

Divide the input, solve smaller instances, and combine the answers:

$$T(n) = \sum_{i=1}^k T(n_i) + f(n).$$

For a equally sized subproblems this becomes

$$T(n) = aT(n/b) + f(n).$$

- a : recursive calls executed; b : size-reduction factor.
- $f(n)$: work outside recursion, including division and combination.
- Fibonacci does not fit: its sizes are $n - 1$ and $n - 2$.

We first assume n is a power of b . Restore floors and ceilings when they affect the proof or algorithm.

Unroll until a base case is reached

1. Write the recurrence and its base cases.
2. Expand the recursive term at least twice.
3. Express the result after an arbitrary k expansions.
4. Find the value of k that reaches a base case.
5. Substitute that value and simplify the finite sums.

Two or three expansions reveal a pattern. The general expression and stopping condition are essential to the argument.

Constant work at each recursive level

Let $c, d > 0$, with n a power of two:

$$T(1) = d, \quad T(n) = T(n/2) + c \quad (n > 1).$$

$$\begin{aligned} T(n) &= c + T(n/2) \\ &= c + (c + T(n/2^2)) \\ &= 2c + T(n/2^2) \\ &= 3c + T(n/2^3). \end{aligned}$$

Each expansion contributes c and halves the remaining argument:

$$T(n) = kc + T(n/2^k).$$

The stopping condition produces a logarithm

Starting from $T(n) = kc + T(n/2^k)$, stop when

$$\frac{n}{2^k} = 1 \iff 2^k = n \iff k = \log_2 n.$$

Hence

$$T(n) = c \log_2 n + T(1) = c \log_2 n + d = \Theta(\log n).$$

The logarithm counts how many halvings reduce the size to one.

All fixed logarithm bases differ by a constant factor.

Nine subproblems produce a geometric sum

Assume n is a power of three:

$$T(1) = 1, \quad T(n) = 9T(n/3) + n \quad (n > 1).$$

$$\begin{aligned} T(n) &= 9(9T(n/3^2) + n/3) + n \\ &= 9^2T(n/3^2) + 3n + n \\ &= 9^3T(n/3^3) + 3^2n + 3n + n. \end{aligned}$$

After k expansions,

$$T(n) = 9^k T(n/3^k) + n \sum_{j=0}^{k-1} 3^j.$$

Evaluate the leaves and the finite sum

The stopping condition $n/3^k = 1$ gives $k = \log_3 n$.

$$9^k = 9^{\log_3 n} = (3^2)^{\log_3 n} = n^2, \quad \sum_{j=0}^{k-1} 3^j = \frac{3^k - 1}{3 - 1} = \frac{n - 1}{2}.$$

Since $T(1) = 1$,

$$T(n) = n^2 + n \frac{n-1}{2} = \frac{3n^2 - n}{2} = \Theta(n^2).$$

Iteration identifies where work accumulates and suggests bounds to prove by substitution.

Recursion Trees

A node represents a call; an edge represents a recursive invocation. A node cost excludes the work of its descendants.

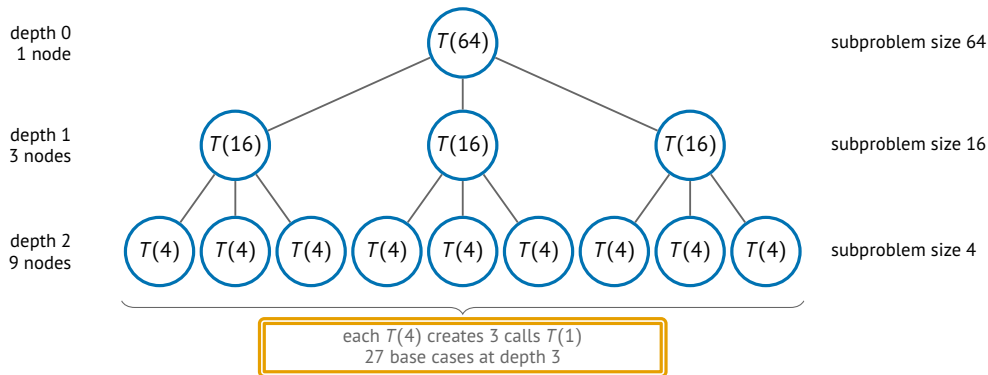
$$T(n) = aT(n/b) + f(n).$$

At depth i	Quantity
Number of nodes	a^i
Subproblem size	n/b^i
Local work per node	$f(n/b^i)$
Total local work	$a^i f(n/b^i)$

The cost of one node is insufficient: the node count changes with depth.

Three calls, each one quarter of the size

$$T(n) = 3T(n/4) + n^2, T(1) = 1, \text{ with } n = 64.$$



Nodes are calls; their arguments give the sizes.

Compute the total work at each depth

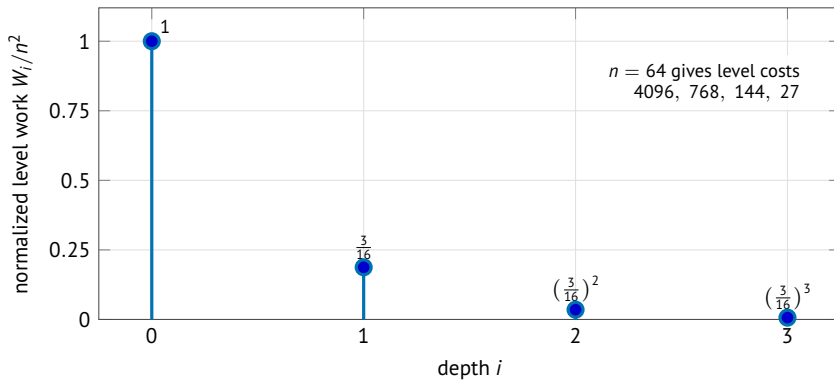
For $T(n) = 3T(n/4) + n^2$, there are 3^i nodes of size $n/4^i$.

$$W_i = 3^i \left(\frac{n}{4^i} \right)^2 = n^2 \left(\frac{3}{16} \right)^i.$$

Although the node count triples, the total work falls by a factor $3/16$ per depth.

The next plot shows W_i/n^2 , separating cost from tree structure.

Level work decreases geometrically



Sum internal work and base cases separately

For powers of four, the depth is $k = \log_4 n$:

$$T(n) = n^2 \sum_{i=0}^{k-1} \left(\frac{3}{16}\right)^i + 3^k T(1).$$

$$\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i = \frac{16}{13}, \quad 3^k = n^{\log_4 3} = O(n^2).$$

The internal work and leaves are both $O(n^2)$. The root alone supplies $\Omega(n^2)$, hence $T(n) = \Theta(n^2)$.

Other patterns

For $2T(n/2) + n$, every depth costs n : total $\Theta(n \log n)$.

If depth costs increase by a fixed factor, the leaves dominate.

The balanced recurrence behind the Master Theorem

$$T(n) = aT(n/b) + f(n), \quad a \geq 1, \quad b > 1,$$

with f eventually non-negative.

Here a, b are constants; write $d = \log_b a$.

Assume n is an exact power of b , recursion stops at size 1, and a base case costs $\Theta(1)$.

- a counts executed calls; b is the reduction factor.
- $f(n)$ includes dividing the instance and combining solutions.

The critical function counts the leaves

Leaves occur when $n/b^h = 1$, so

$$h = \log_b n, \quad a^h = a^{\log_b n} = n^{\log_b a}.$$

Definition (Critical function)

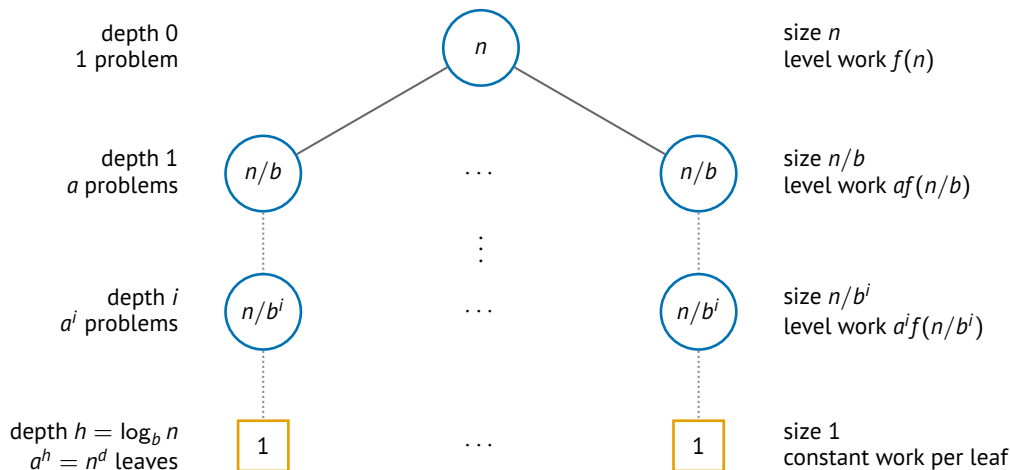
For $T(n) = aT(n/b) + f(n)$, define

$$d = \log_b a, \quad g(n) = n^d = n^{\log_b a}.$$

The function $g(n)$ measures the number of leaves, up to the constant cost of each base case.

First compute d and $g(n)$; then compare root work $f(n)$ with leaf work $g(n)$.

Read sizes and costs from the tree



The Master Theorem when the leaves matter

Let $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$ are constants and f is eventually non-negative. Set $d = \log_b a$. Use powers of b and constant positive base costs as above.

Case 1

If $f(n) = O(n^{d-\varepsilon})$ for some $\varepsilon > 0$, then

$$T(n) = \Theta(n^d).$$

Case 2

If $f(n) = \Theta(n^d)$, then

$$T(n) = \Theta(n^d \log n).$$

Case 1 is leaf-dominated; in Case 2, all depths contribute comparably.

The Master Theorem when the root matters

For $T(n) = aT(n/b) + f(n)$, keep the same hypotheses and $d = \log_b a$.

Case 3

If $f(n) = \Omega(n^{d+\varepsilon})$ for some $\varepsilon > 0$ and

$$af(n/b) \leq cf(n)$$

for some constant $c < 1$ and all sufficiently large n , then

$$T(n) = \Theta(f(n)).$$

Cases 1 and 3 require a **polynomial gap**; a logarithmic difference is insufficient.

Regularity bounds the next depth by a fixed fraction of the root cost. Repeating it gives geometric decrease.

How to read the three cases

The Master theorem compares two sources of work:

work near the leaves $\Theta(n^d)$ and work introduced at the root $f(n)$.

- If $f(n)$ is polynomially smaller than n^d , the leaves dominate.
- If $f(n)$ has the same order as n^d , every level contributes equally.
- If $f(n)$ is polynomially larger than n^d , the root dominates, provided the regularity condition prevents lower levels from becoming large again.

Why the tree yields the three cases

Proof sketch. For $h = \log_b n$,

$$T(n) = \underbrace{a^h T(1)}_{\Theta(n^d)} + \sum_{i=0}^{h-1} a^i f(n/b^i).$$

- **Case 1:** the hypothesis bounds depth work by a growing geometric sequence. Its sum is $O(n^d)$; leaves give the matching lower bound.
- **Case 2:** each depth contributes $\Theta(n^d)$; there are $\Theta(\log n)$ depths.
- **Case 3:** regularity gives geometric decrease from the root. The root supplies a matching lower bound.

Clarification: in Case 1, the geometric growth describes an upper bound; actual depth costs need not grow monotonically.

Apply the theorem in a fixed order

1. Write $T(n) = aT(n/b) + f(n)$.
2. Check constant $a \geq 1$, $b > 1$, and eventual non-negativity of f .
3. Compute $d = \log_b a$ and $g(n) = n^d$.
4. Compare $f(n)$ with $g(n)$; check the required polynomial gap.
5. In Case 3, verify $af(n/b) \leq cf(n)$ for a constant $c < 1$.
6. Name the case and state its bound.

Frequent errors

Comparing with n^a or n^b ; counting regions instead of calls; using a strict inequality without a polynomial gap; omitting regularity.

Two applications of Case 2

Example 1: Constant root work

$$T(n) = T(n/2) + \Theta(1): a = 1, b = 2, f(n) = \Theta(1).$$

$$d = \log_2 1 = 0, \quad g(n) = 1, \quad T(n) = \Theta(\log n).$$

The logarithm counts successive halvings.

Example 2: Linear root work

$$T(n) = 2T(n/2) + \Theta(n): a = 2, b = 2, f(n) = \Theta(n).$$

$$d = 1, \quad g(n) = n, \quad T(n) = \Theta(n \log n).$$

Every depth contains $\Theta(n)$ merge work.

Linear local work can create quadratic total work

$$T(n) = 9T(n/3) + n.$$

Here $a = 9$, $b = 3$, $f(n) = n$, and

$$d = \log_3 9 = 2, \quad g(n) = n^2.$$

Since $n = O(n^{2-1})$, Case 1 applies with $\varepsilon = 1$:

$$T(n) = \Theta(n^2).$$

One call performs linear work, but branching creates n^2 leaves.

Verify regularity before choosing Case 3

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

Here $a = 3$, $b = 2$, $d = \log_2 3 < 2$, and $g(n) = n^d$. The polynomial gap is $\varepsilon = 2 - d > 0$.

$$3f(n/2) = 3\left(\frac{n}{2}\right)^2 = \frac{3}{4}f(n).$$

Thus regularity holds with $c = 3/4 < 1$, and Case 3 gives

$$T(n) = \Theta(n^2).$$

Total local work decreases by a factor $3/4$ at each depth.

A logarithmic gap lies outside the classical cases

$$T(n) = 2T(n/2) + n \log n, \quad d = 1, \quad g(n) = n.$$

The work $n \log n$ exceeds n but is not polynomially larger: $\log n / n^\varepsilon \rightarrow 0$ for every fixed $\varepsilon > 0$.

Extended second case

If $f(n) = \Theta(n^d \log^k n)$ for a fixed $k \geq 0$, then

$$T(n) = \Theta(n^d \log^{k+1} n).$$

The classical case has $k = 0$; here $k = 1$ gives $\Theta(n \log^2 n)$. A recursion-tree sum gives the same result.

Recognize when another method is needed

The theorem does not apply directly when

- subproblem sizes differ, as in $T(n - 1) + T(n - 2)$;
- a or b depends on n ;
- the recurrence has the right shape but none of the stated cases applies.

Use iteration, substitution, or a direct recursion-tree analysis. A failed applicability test does not imply that the recurrence has no useful bound.

Use sorted order to discard candidates

Input and result

A sorted array A with constant-time random access, and a search key. Return the index of its *first occurrence*, or Not-found.

The interval still to inspect is half-open: $[low, high)$. It includes low , excludes $high$, and contains $high - low$ positions.

A midpoint comparison narrows the search for the first position whose value is at least the key.

- Without sortedness, the discarded half cannot be ruled out.
- Locating a midpoint in a linked list can itself take linear time.

Binary search for the first occurrence

```
Binary-search(A, key)
  low <- 0
  high <- length(A)
  while low < high
    middle <- low + floor((high-low)/2)
    if A[middle] < key
      low <- middle + 1
    else
      high <- middle
  if low < length(A) and A[low] = key
    return low
  return Not-found
```

The final test checks the index before reading the array: evaluate and from left to right, stopping if the first condition is false.

Maintain ordered regions around the search interval

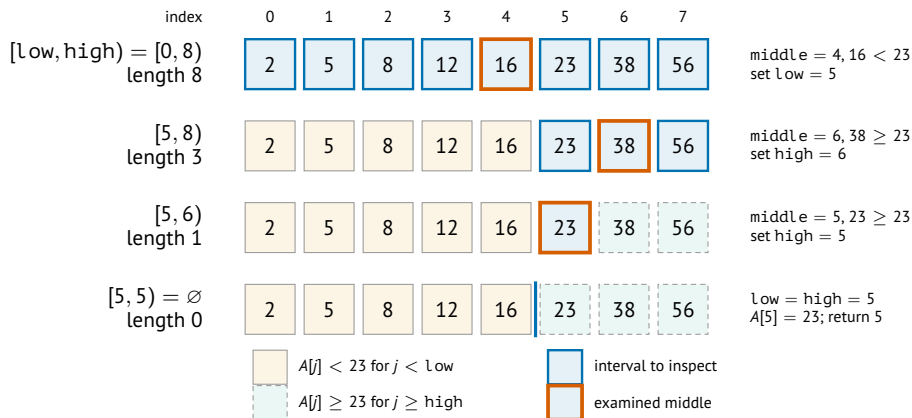
Throughout the loop, for valid array indices $0 \leq j < \text{length}(A)$,

$$0 \leq \text{low} \leq \text{high} \leq \text{length}(A),$$

$$A[j] < \text{key} \ (j < \text{low}), \quad A[j] \geq \text{key} \ (j \geq \text{high}).$$

- If $A[\text{middle}] < \text{key}$, sortedness justifies setting $\text{low} = \text{middle} + 1$.
- Otherwise, set $\text{high} = \text{middle}$. Keep that position as the right boundary and search to its left, even on equality.
- When the interval is empty, check that low is in range and its value equals the key.

Finding the first position of 23



Every iteration makes the interval smaller

Let $n = \text{high} - \text{low} > 0$. The midpoint is

$$\text{middle} = \text{low} + \lfloor n/2 \rfloor.$$

Size after the update

- Set $\text{high} = \text{middle}$: the new size is $\lfloor n/2 \rfloor$.
- Set $\text{low} = \text{middle} + 1$: the new size is $n - \lfloor n/2 \rfloor - 1$.

Both sizes are at most $\lfloor n/2 \rfloor < n$.

Thus the loop terminates. A one-position interval becomes empty after one iteration; an initially empty array skips the loop.

Binary search has logarithmic time and constant iterative space

Let n be the current interval length. An iteration costs $\Theta(1)$ and leaves at most $\lfloor n/2 \rfloor$ positions to inspect.

Worst-case running time

$$T(0) = \Theta(1), \quad T(n) = T(\lfloor n/2 \rfloor) + \Theta(1) \quad (n \geq 1).$$

Repeated halving gives $\Theta(\log n)$ time as n grows. Including empty inputs, write $\Theta(1 + \log(n + 1))$.

- The displayed iterative code uses $\Theta(1)$ auxiliary space.
- A recursive version uses $\Theta(\log n)$ stack space in the worst case.

Exercises in iteration and substitution

Exercise 1

Solve

$$T(n) = T(n/3) + n, \quad T(1) = 1,$$

by iteration for powers of three. Display the first three expansions, derive the expression after k expansions, and evaluate the resulting geometric sum.

Exercise 2

Draw the recursion tree for

$$T(n) = 4T(n/2) + n.$$

For depth i , state the number of nodes, the subproblem size, the local work per node, and the total work. Sum all depths and verify the result using the Master Theorem.

Exercise 3

Trace binary search on $[1, 2, 2, 2, 5, 8]$ for the keys 2, 4, and 9. Record low, high, and middle at every iteration. Verify the invariant after each update.

Exercises on the limits of the Master Theorem

Exercise 4

Apply the Master Theorem to each recurrence for which it is applicable. For every application, identify a , b , $f(n)$, $d = \log_b a$, the relevant case, and every additional condition that must be checked.

$$T_1(n) = 16T_1(n/4) + n,$$

$$T_2(n) = 4T_2(n/2) + n^2,$$

$$T_3(n) = T_3(n/2) + 2^n,$$

$$T_4(n) = 2T_4(n/2) + n \log n.$$

For any recurrence outside the theorem, explain precisely which hypothesis fails.

Exercise 6

Solve

$$T(n) = 2T(n/2) + n \log n$$

using a recursion tree. Compute the work at depth i before summing the depths.

Reading pointers

- CLRS, *Introduction to Algorithms*, 3rd edition, Section 4.3 and recurrence analysis in Chapter 4.
- Kleinberg and Tardos, *Algorithm Design*, Sections 5.1–5.2.
- Dasgupta, Papadimitriou, and Vazirani, Sections 2.2–2.3 and the binary-search box.

Thank you!



UNIVERSITY
OF TRIESTE