# CS 5413
# Chapter 2

Getting Started

**Overview and Goals:**

- Describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- Algorithms in pseudocode.
- Asymptotic notation to express running-time analysis.
- “Divide and conquer" in the context of merge sort.

**The sorting problem**

**Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

**Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The sequences are typically stored in arrays.

We will see several ways to solve the sorting problem.
Each way will be expressed as an ***algorithm***: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

**Expressing algorithms**

We express algorithms in whatever way is the clearest and most concise.

When issues of control need to be made perfectly clear, we often use ***pseudocode***.

Pseudocode is designed for expressing algorithms to humans. Software engineering issues of data abstraction, modularity, and error handling are often ignored.

We sometimes embed English statements into pseudocode. Therefore, unlike for "real" programming languages, we cannot create a compiler that translates pseudocode to machine code.

**Insertion sort**

A good algorithm for sorting a small number of elements. It works the way you might sort a hand of playing cards:

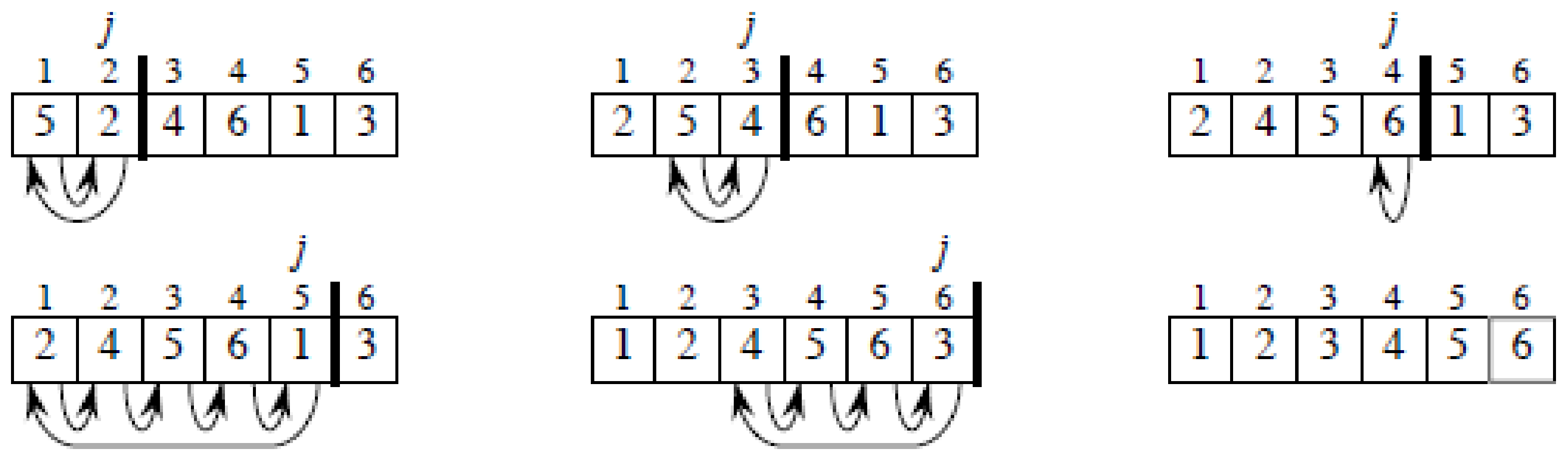
- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

***Pseudocode:*** We use a procedure INSERTION-SORT.

- Takes as parameters an array A[l .. n] and the length n of the array.
- We use ". ." to denote a range within an array.
- The array A is sorted in place: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

```
INSERTION-SORT(A)
for j ← 2 to n
    do key ← A[j]
       ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

*Example:*


j
1 2 3 4 5 6
5 2 4 6 1 3
j
1 2 3 4 5 6
2 5 4 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
2 4 5 6 1 3
j
1 2 3 4 5 6
1 2 4 5 6 3
1 2 3 4 5 6
1 2 3 4 5 6

**Correctness**
We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer. Here's the loop invariant for INSERTION-SORT:

**Loop invariant:** At the start of each iteration of the "outer" **for** loop
- the loop indexed by $j$
- the subarray $A[1 \,..\, j-1]$ consists of the elements originally in $A[1 \,..\, j-1]$ but in sorted order.

To use a loop invariant to prove correctness, we must show three things about it:

**Initialization:** It is true prior to the first iteration of the loop.
**Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
**Termination:** When the loop terminates, the invariant-usually along with the reason that the loop terminated gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like **mathematical induction**:

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.

***For insertion sort:***

**Initialization:** Just before the first iteration, $j = 2$. The subarray $A[1 .. j - 1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

**Maintenance:** To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.

**Termination:** The outer **for** loop ends when $j > n$; this occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1 .. n]$ consists of the elements originally in $A[1 .. n]$ but in sorted order. In other words, the entire array is sorted!

**Analyzing algorithms**

We want to predict the resources that the algorithm requires. Usually, running time.

In order to predict resource requirements, we need a computational model.

**How do we analyze an algorithm's running time?**

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

***Input size:***

Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

***Running time:***

On a particular input, it is the number of primitive operations (steps) executed.

- Want to define steps to be machine-independent.
- Figure that each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line i takes the same amount of time $c_i$. This is assuming that the line consists only of primitive operations.
    - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
    - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: "sort the points by x -coordinate."

**Analysis of insertion sort**

- Assume that the i th line takes time $c_i$ , which is a constant. (Since the third line is a comment, it takes no time.)
- For j = 2, 3,... , n, let $t_j$ be the number of times that the while loop test is executed for that value of j .
- Note that when a for or while loop exits in the usual way due to the test in the loop header the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Let *T (n)* = running time of I insertion sort T .

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) . \end{aligned}$$

The running time depends on the values of $t_j$ . These vary according to the input.

*Best case:* The array is already sorted.

- Always find that $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All $t_j$ are 1.
- Running time is

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}
$$

- Can express $T(n)$ as $an + b$ for constants $a$ and $b$ (that depend on the statement costs $c_i$) $\Rightarrow T(n)$ is a *linear function* of $n$.

*Worst case:* The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the $j$th position $\Rightarrow$ compare with $j-1$ elements.
- Since the while loop exits because $i$ reaches 0, there's one additional test after the $j-1$ tests $\Rightarrow t_j = j$.
- $\displaystyle\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\displaystyle\sum_{j=2}^{n}(t_j-1) = \sum_{j=2}^{n}(j-1)$.
- $\displaystyle\sum_{j=1}^{n} j$ is known as an *arithmetic series*, and equation (A.1) shows that it equals $\dfrac{n(n+1)}{2}$.
- Since $\displaystyle\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\dfrac{n(n+1)}{2} - 1$.
- Letting $k = j-1$, we see that $\displaystyle\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants $a, b, c$ (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of $n$.

- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants $a, b, c$ (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of $n$.

**Worst-case and average-case analysis**

We usually concentrate on finding the ***worst-case running time***: the longest running time for *any* input of size $n$.

***Reasons:***

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it's often about as bad as the worst case.

***Example:***

Suppose that we randomly choose $n$ numbers as the input to insertion sort.

On average, the key in $A[\,j\,]$ is less than half the elements in $A[l\,..\,j-l]$ and it's greater than the other half.

⇒ On average, the **while** loop has to look halfway through the sorted subarray $A[1\,..\,j-l]$ to decide where to drop *key*.

⇒ $t_j = j/2$.

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of $n$.

**Order of growth**

Another abstraction to ease analysis and focus on the important features.
Look only at the leading term of the formula for running time.

- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

***Example:***

For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.

Drop lower-order terms ⇒ $an^2$.

Ignore constant coeficient ⇒ $n^2$.

But we cannot say that the worst-case running time *T (n) equals* $n^2$. It grows like $n^2$. But it doesn't equal $n^2$. We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is $n^2$.

We usually consider one algorithm to be more efficient than another if its worst- case running time has a smaller order of growth.

**Designing algorithms**

There are many ways to design algorithms.

For example, insertion sort is ***incremental***: having sorted $A[1 \,..\, j-1]$, place $A[j]$ correctly, so that $A[1 \,..\, j]$ is sorted.

**Divide and conquer**
Another common approach.

**Divide** the problem into a number of subproblems.
**Conquer** the subproblems by solving them recursively.
***Base case:*** If the subproblems are small enough, just solve them by brute force.
**Combine** the subproblem solutions to give a solution to the original problem.

**Merge sort**

A sorting algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.
Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.
To sort $A[p .. r]$:

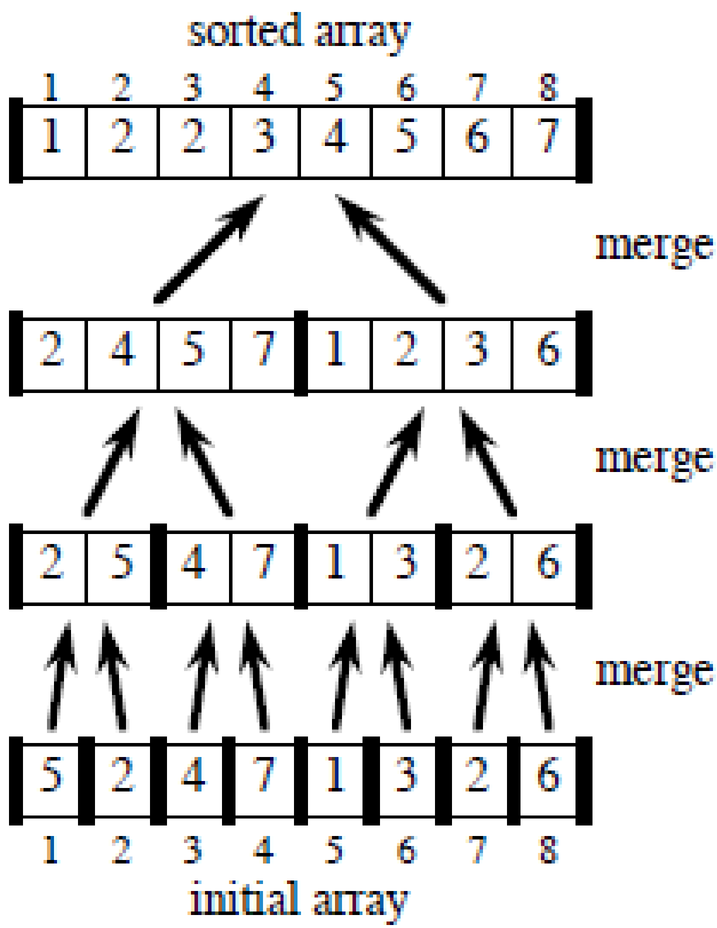
> **Divide** by splitting into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, where $q$ is the halfway point of $A[p .. r]$.
> **Conquer** by recursively sorting the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.
> **Combine** by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ to produce a single sorted subarray $A[p .. r]$. To accomplish this step, we'll define a procedure MERGE$(A, p, q, r)$.

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

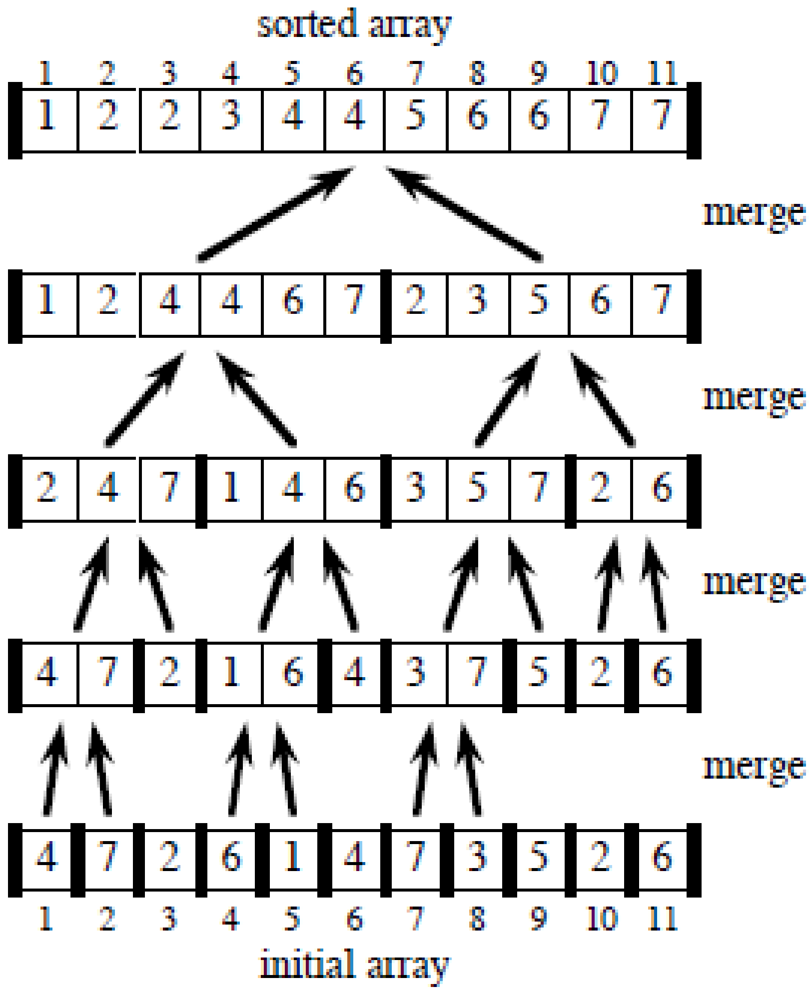
```
MERGE-SORT(A, p, r)
if p < r
  then q ← ⌊(p + r)/2⌋
       MERGE-SORT(A, p, q)
       MERGE-SORT(A, q + 1, r)
       MERGE(A, p, q, r)
```

*Initial call:* MERGE-SORT$(A, 1, n)$

***Example:*** Bottom-up view for $n$ = 8:



Bottom-up view for $n = 11$:



**Merging**

What remains is the MERGE procedure.

**Input:** Array *A* and indices *p, q, r* such that

- $p \leq q < r$ .
- Subarray A[ p .. q] is sorted and subarray A[q + 1 .. r ] is sorted. By the restrictions on p, q, r , neither subarray is empty.

**Output:** The two subarrays are merged into a single sorted subarray in *A*[ *p* .. *r* ].

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$ = the number of elements being merged.

***What is n?*** Until now, *n* has stood for the size of the original problem. But now we're using it as the size of a subproblem. We will use this technique when we analyze recursive algorithms. Although we may denote the original problem size by *n*, in general *n* will be the size of a given subproblem.

*Pseudocode:*

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
     do L[i] ← A[p + i − 1]
for j ← 1 to n2
     do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
     do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

*Example:* A call of MERGE(9, 12, 16)

***Running time:***

The first two **for** loops take Θ(n1 + n2) = Θ(n) time. The last for loop makes n iterations, each taking constant time, for Θ(n) time. Total time: Θ(n).

## Analyzing divide-and-conquer algorithms

Use a ***recurrence equation*** (more commonly, a ***recurrence***) to describe the running time of a divide-and-conquer algorithm.
Let $T(n)$ = running time on a problem of size $n$.

- If the problem size is small enough (say, $n \le c$ for some constant $c$), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into $a$ subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size-n problem be $D(n)$.
- There are a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

**Analyzing merge sort**

For simplicity, assume that $n$ is a power of 2 ⇒ each divide step yields two subproblems, both of size exactly $n/2$. The base case occurs when $n = 1$. When $n \geq 2$, time for merge sort steps:

**Divide:** Just compute $q$ as the average of $p$ and $r$ ⇒ $D(n) = \Theta(1)$.

**Conquer:** Recursively solve 2 subproblems, each of size $n/2$ ⇒ $2T(n/2)$.

**Combine:** MERGE on an $n$-element subarray takes $\Theta(n)$ time ⇒ $C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in $n$: $\Theta(n)$ ⇒ recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

***Solving the merge-sort recurrence:***

By the **master theorem** in Chapter 4, we can show that this recurrence has the solution *T (n) = Θ(n* lg *n)*. [Reminder: lg *n* stands for $\log_2 n$.]
Compared to insertion sort (*Θ(n²)* worst-case time), merge sort is faster. Trading a factor of *n* for a factor of *lg n* is a good deal.
On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

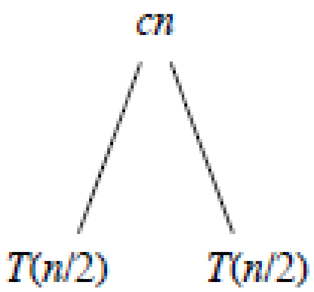
We can understand how to solve the merge-sort recurrence **without the master theorem**.

- Let *c* be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps.
- We rewrite the recurrence as

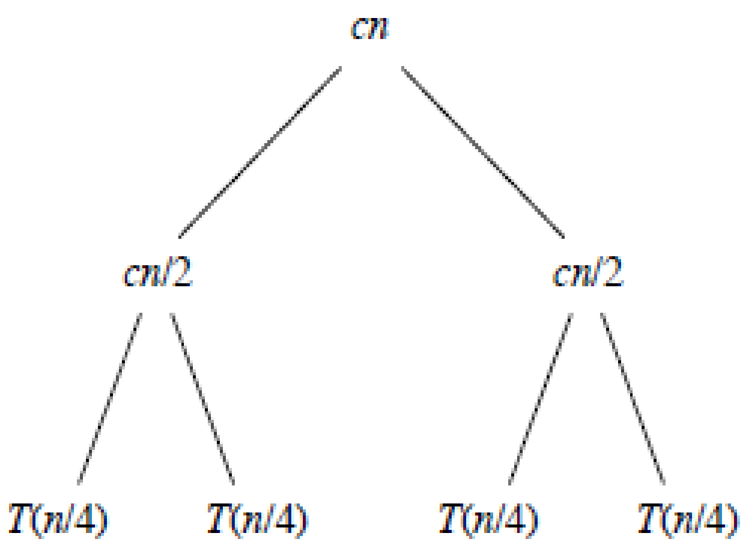
$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,. \end{cases}$$

Draw a *recursion tree*, which shows successive expansions of the recurrence.

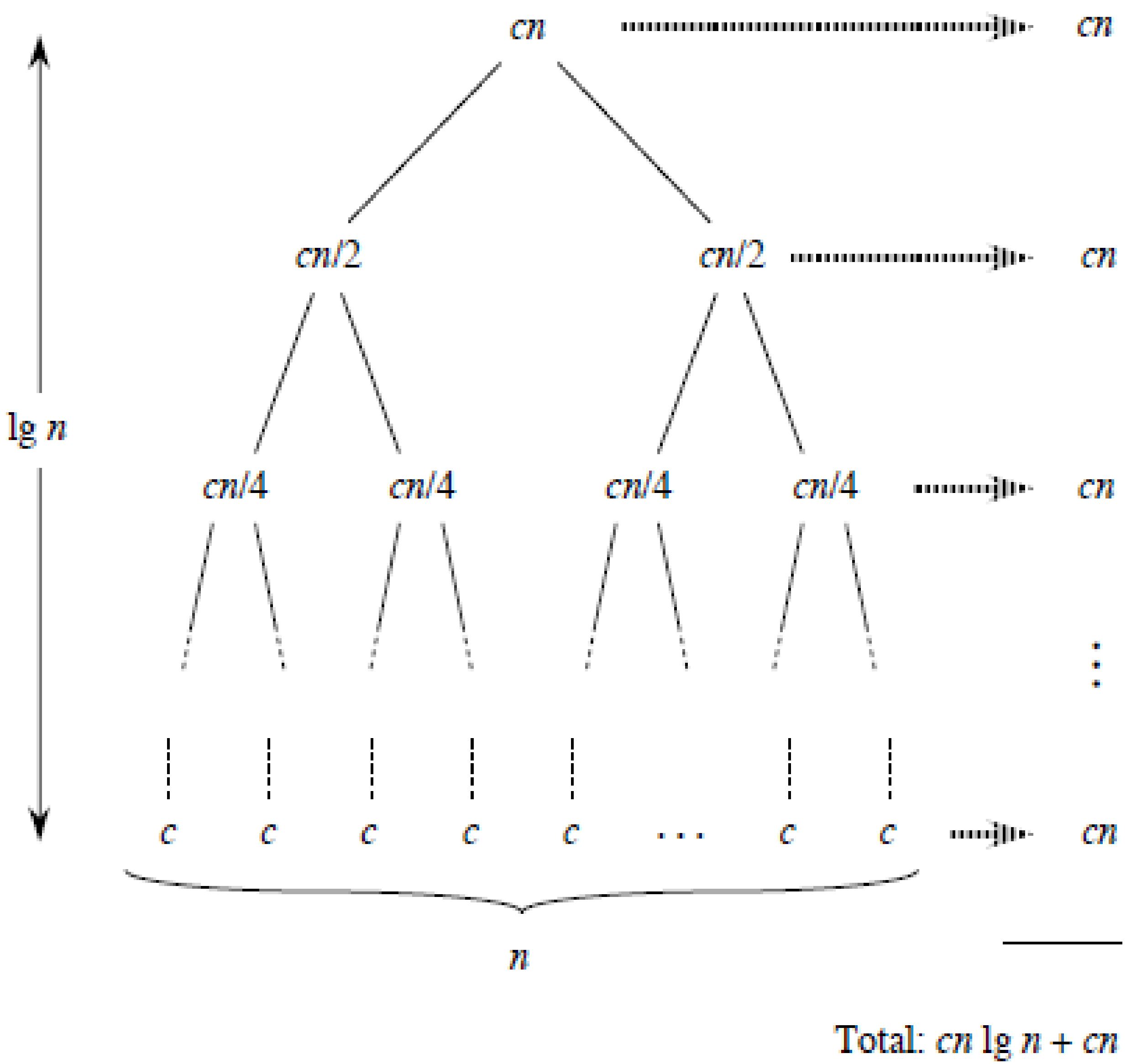
For the original problem, we have a cost of $cn$, plus the two subproblems, each costing $T(n/2)$:




For each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:



Continue expanding until the problem sizes get down to 1:



Each level has cost *cn*.

- The top level has cost *cn*.
- The next level down has 2 subproblems, each contributing cost *cn/2*.
- The next level has 4 subproblems, each contributing cost *cn/4*.
- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves ⇒ cost per level stays the same.

There are $lg\ n + 1$ levels (height is $lg\ n$).

- Use induction.
- Base case: n = 1 ⇒ 1 level, and lg 1 + 1 = 0 + 1 = 1.
- Inductive hypothesis is that a tree for a problem size of $2^i$ has $lg\ 2^i\ +1 = i +1$ levels.
- Because we assume that the problem size is a power of *2*, the next problem size up after $2^i$ is $2^{i+1}$ .
- A tree for a problem size of $2^{i+1}$ has one more level than the size-$2^i$ tree ⇒ $i + 2$ levels.
- Since $lg\ 2^{i+1} + 1 = i + 2$, we're done with the inductive argument.

Total cost is sum of costs at each level. Have $lg\ n + 1$ levels, each costing *cn* ⇒
total cost is $cn\ lg\ n + cn$.

Ignore low-order term of en and constant coeficient c ⇒ *Θ* >(n lg n).