CSE 100: Algorithm Design and Analysis
Chapter 02: Getting Started

Sungjin Im

University of California, Merced

Last Update: 01-26-2023

Meaning is not something you stumble across, like the answer to a riddle or the prize in a treasure hunt. Meaning is something you build into your life. You build it out of your own past, out of your affections and loyalties, ..., out of the values for which you are willing to sacrifice something. The ingredients are there. You are the only one who can put them together into that unique pattern that will be your life. Let it be a life that has dignity and meaning for you. If it does, then the particular balance of success or failure is of less account.

by John W. Gardner

Outline

- Proving correctness using *loop invariant* and/or *induction*.
- Intro of a computation model, *RAM* (Random-Access Machine).
- *Worst case* vs average case running time.
- A glimpse at divide-and-conquer (merge-sort) and its runtime analysis.

Review: The Sorting Problem

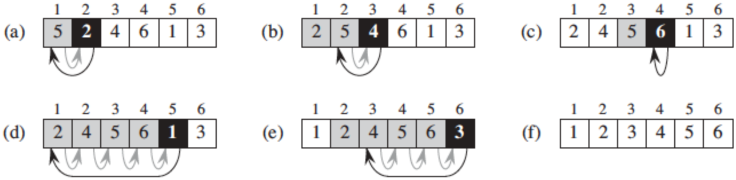
Input: A sequence of n numbers/elements $\langle a_1, a_2, \cdots, a_n \rangle$.
Output: A permutation (reordering) $\langle a'_1, a'_2, \cdots, a'_n \rangle$ of the input sequence s.t. $a'_1 \leq a'_2 \leq a'_3 \leq \cdots \leq a'_n$.

Review: Insertion sort

Input: $A[1 \cdots n]$

In the jth iteration ($2 \leq j \leq n$), we ensure that $A[1 \cdots j]$ is sorted by inserting $A[j]$, the jth number in the the 'right' position of $A[1 \cdots j-1]$. Here we do so by reverse-scanning the array $A[1 \cdots j-1]$ *sequentially* and pushing back elements therein greater than the jth element.

(Remark: You can abstract something if it doesn't affect the asymptotic running time of the algorithm.)



Review: Insertion sort

Input: $A[1 \cdots n]$

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
          sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Review: Insertion sort

```
Input: A[1 ... n]
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Insertion sort

Show correctness via loop invariant

Loop Invariant: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
Input: A[1 ... n]
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Insertion sort

Show correctness via loop invariant

- 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 gives us a useful property that helps show that the algorithm is correct.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Initialization: It is true prior to the first iteration of the loop.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Initialization: It is true prior to the first iteration of the loop.

- Just at the start of the first iteration ($j = 2$), $A[1]$ is ordered, and the number was originally in $A[1]$.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

- Say the invariant holds true for iter' $j \geq 2$.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

- Say the invariant holds true for iter' $j \geq 2$.
- At the end of the iter', we have $A[1...i]$, $A[j]$, and $A[i+1...j-1]$ in this order in a prefix of A, which we want show is sorted.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

- Say the invariant holds true for iter' $j \geq 2$.
- At the end of the iter', we have $A[1...i]$, $A[j]$, and $A[i+1...j-1]$ in this order in a prefix of A, which we want show is sorted.
- We know that $A[1...i]$ and $A[i+1...j-1]$ are sorted by the invariant.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

- Say the invariant holds true for iter' $j \geq 2$.
- At the end of the iter', we have $A[1...i]$, $A[j]$, and $A[i+1...j-1]$ in this order in a prefix of A, which we want show is sorted.
- We know that $A[1...i]$ and $A[i+1...j-1]$ are sorted by the invariant.
- Further, $A[i] \leq A[j] \leq A[i+1]$ by Algo's def, meaning $A[1...j]$ is sorted at the end of iter' as desired.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

- Say the invariant holds true for iter' $j \geq 2$.
- At the end of the iter', we have $A[1...i]$, $A[j]$, and $A[i+1...j-1]$ in this order in a prefix of A, which we want show is sorted.
- We know that $A[1...i]$ and $A[i+1...j-1]$ are sorted by the invariant.
- Further, $A[i] \leq A[j] \leq A[i+1]$ by Algo's def, meaning $A[1...j]$ is sorted at the end of iter' as desired.
- Clearly, all elements in $A[1...j]$ originate from the same subarray.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Maintenance:

- Say the invariant holds true for iter' $j \geq 2$.
- At the end of the iter', we have $A[1...i]$, $A[j]$, and $A[i+1...j-1]$ in this order in a prefix of A, which we want show is sorted.
- We know that $A[1...i]$ and $A[i+1...j-1]$ are sorted by the invariant.
- Further, $A[i] \leq A[j] \leq A[i+1]$ by Algo's def, meaning $A[1...j]$ is sorted at the end of iter' as desired.
- Clearly, all elements in $A[1...j]$ originate from the same subarray.
- Thus, the loop invariant holds true at the start of the next iter' $j+1$.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
Input: A[1 ... n]
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion sort

Show correctness via loop invariant

LI: At the start of the jth iter' of the for loop, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$, but in sorted order.

```
Input: A[1 ... n]
for j = 2 to A.length
  Find 1 <= i <= j-1 s.t. A[i] <= A[j] <= A[i+1] if such i
     exists; otherwise, i = 0
  Push A[i+1 ... j-1] to A[i ... j]
  while moving A[j] to A[i+1] (need to use a temp var.)
```

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

- The for loop ends when $j = n + 1$, and the loop invariant implies that the array is sorted as desired.

Analysis of Algorithms

- Analyzing an algorithm means predicting the resources that the algorithm requires. Mainly running time.
- A formal model is needed. cf. running time using 1 processor vs. thousands of processors.

Random-access machine (RAM) model

- Our ‘default’ computation model.
- Single processor: Instructions are executed sequentially. No concurrent operations are allowed.
- Basic operations such as addition, multiplication, load, store, copy, control, initialization are assumed to take a constant amount of time each (if numbers are big, this may not be true, but we assume that this is the case unless stated otherwise).
- Simple random-access (no hierarchy).

Running time parameterized by input size

How to measure input size

- The most precise measure is # of bits used to express the input.
- In practice, it is often # of elements/items in the input.
- Sometimes, several parameters are used. For example, # of vertices and # of edges are used to measure graph sizes.

Usefulness of asymptotic running times

- Some operations may take more time.
- Exact counting is a huge pain.
- We care about efficiency when the input is large.

Quick review of asymptotic running time notations

- $O(f)$: at most f within a constant factor.
- $\Omega(f)$: at least f within a constant factor.
- $\Theta(f)$: if is $O(f)$ and $\Omega(f)$ simultaneously.

Asymptotic running times

Convention

If you're asked what is the running time of an algorithm, you're expected to say that it is $O(\cdot)$. But you want to state it as tight as possible.

Worst case vs. Average case

For each input, the running time is the number of operations (machine-indep.) executed. When the input is parameterized by its size,

- Worst case: $T(n)$ is the maximum running time for *any* input of size n.
- Average case: $T(n)$ is the average running time for inputs of sizes n.

In this course, we will focus on the worst case running time. So in this course, by runtime time we mean the worst case running time unless stated otherwise.

So when we say the running time is $O(n^2)$, we mean that the algorithm performs $O(n^2)$ basic operations for all inputs of size at most n.

Running time of insertion sort

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

Running time of insertion sort

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

$T(n) = \sum_{j=2}^{n} O(j) = O(n^2)$.

Running time of insertion sort

In the jth iteration $(2 \leq j \leq n)$, we ensure that $A[1 \cdots j]$ is sorted by inserting $A[j]$, the jth number in the the 'right' position of $A[1 \cdots j-1]$. Here we do so by reverse-scanning the array $A[1 \cdots j-1]$ *sequentially* and pushing back elements therein greater than the jth element.

Running time of insertion sort

In the jth iteration $(2 \leq j \leq n)$, we ensure that $A[1 \cdots j]$ is sorted by inserting $A[j]$, the jth number in the the 'right' position of $A[1 \cdots j-1]$. Here we do so by reverse-scanning the array $A[1 \cdots j-1]$ *sequentially* and pushing back elements therein greater than the jth element.

$T(n) = O(n) * O(n) = O(n^2)$.

Running time of insertion sort

In the jth iteration ($2 \leq j \leq n$), we ensure that $A[1 \cdots j]$ is sorted by inserting $A[j]$, the jth number in the the 'right' position of $A[1 \cdots j-1]$. Here we do so by reverse-scanning the array $A[1 \cdots j-1]$ *sequentially* and pushing back elements therein greater than the jth element.

$T(n) = O(n) * O(n) = O(n^2)$.

Note:
The analysis is tight. IS takes $\Omega(n^2)$ for the input $\langle n, n-1, ..., 1 \rangle$.

Designing algorithms: incremental vs. divide-and-conquer

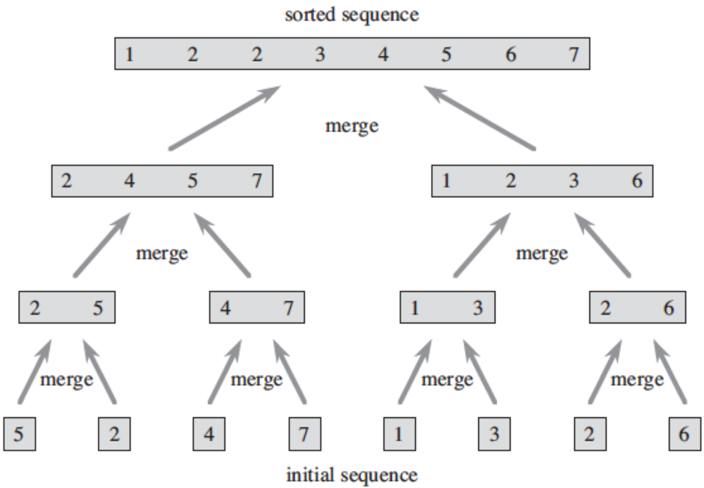
Divide and Conquer

- Divide the problem into a number of smaller subproblems.
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem.

Sorting via Divide and Conquer: Merge sort

- Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- Conquer: Sort the two subsequences recursively using merge sort. If there's only one element, do nothing.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

Merge sort



sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Merge sort

Sort $A[p \ldots r]$

MERGE-SORT(A, p, r)

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

Merge

Merge two sorted arrays $A[p...q]$ and $A[q+1...r]$ into a sorted array $A[p...r]$

Copy $A[p...q]$ to a new temp array $L[1..q-p+1]$.
Copy $A[q+1...r]$ to a new temp array $R[1...r-q]$.
Keep a pointer i for L starting from index 1.
Keep a pointer j for R starting from index 1.
Keep a pointer k for A starting from index p.
Compare $L[i]$ and $R[j]$, and copy the smaller one to $A[k]$. Either $i++$ or $j++$ depending on the element copied, and $k++$.
When all elements are copied to A from L and R, we finish.

Merge

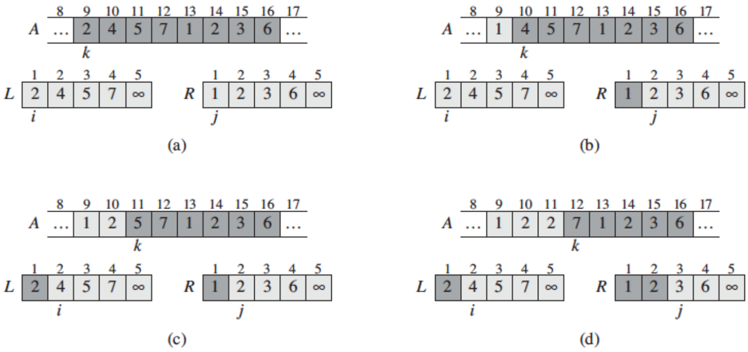
Merge two sorted arrays $A[p...q]$ and $A[q+1...r]$ into a sorted array $A[p...r]$

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

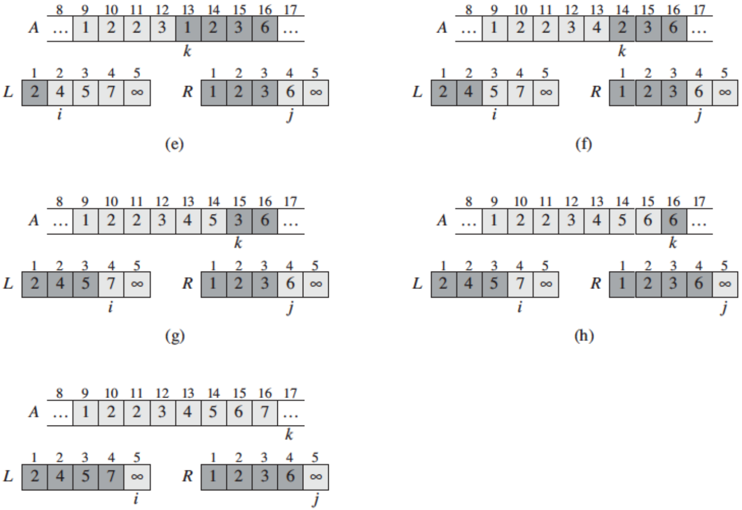
Merge

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Merge



Merge



8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(e)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(f)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(g)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
(h)
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

Merge sort

Correctness

We can show that Merge is correct using a loop invariant (See the textbook and discussion session problems Ch02). Assuming that Merge is correct, we can show Merge sort is correct. We show the correctness by induction on the number of elements.

Base case: $n = 1$. Trivial.

Induction step: Assuming that Merge sort is correct for *all* inputs of size less than n, $(n \geq 2)$, we want to show that it is correct also for all inputs of size n.

Merge sort

Correctness

We can show that Merge is correct using a loop invariant (See the textbook and discussion session problems Ch02). Assuming that Merge is correct, we can show Merge sort is correct. We show the correctness by induction on the number of elements.

Base case: $n = 1$. Trivial.

Induction step: Assuming that Merge sort is correct for *all* inputs of size less than n, $(n \geq 2)$, we want to show that it is correct also for all inputs of size n.

Say, $r - p$ is n. Then, $q - (p - 1), r - q < n$. By induction hypothesis, we know that after lines 3, 4, $A[p...q]$ and $A[q + 1...r]$ are sorted. Then, in Line 5, the two subarrays are merged into $A[p...r]$ and all elements in $A[p...r]$ are sorted.

Merge sort

Running time

Merge sort

Running time

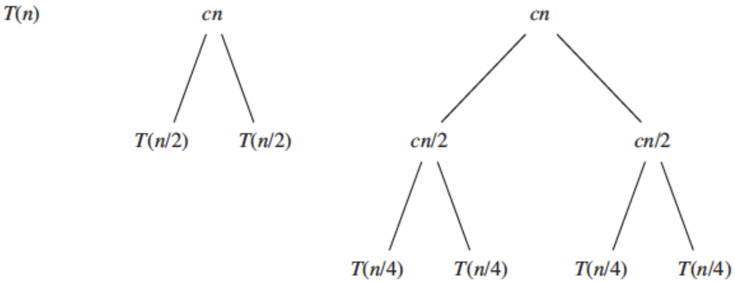
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

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

We will learn how to solve recursions in Chapter 4. Here we will briefly go over recursion tree.

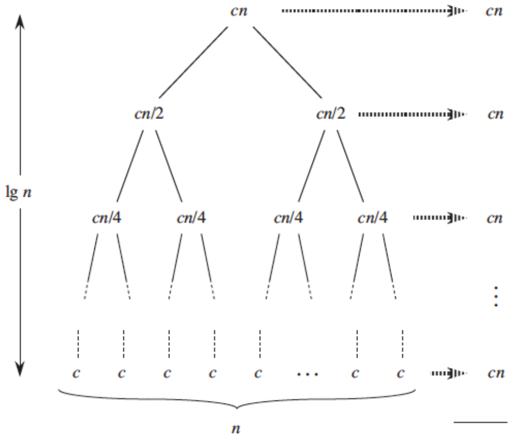
Merge sort

Running time via recursion tree



Merge sort

Running time via recursion tree



(d)