ALGORITHMS – COMTEK3,CCT3 & ESD3
Analysis and Design of Algorithms

Ramoni Adeogun
Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

AALBORG UNIVERSITET

Outline

- Analysis of algorithms
  - Insertion sort
  - Running time analysis
- Correctness of algorithms
- Design of algorithms
  - Divide and conquer
    - Merge sort
    - Merge algorithm
    - Analysis of merge sort
- Summary and key takeaways

Analysis of Algorithms

AALBORG UNIVERSITET

The Sorting Problem

The Problem:
Given an unsorted array $A = [A[1], \cdots, A[n]]$ of n numbers, re-arrange the entries in increasing (or decreasing) order.

Precondition P_1:

- n: a positive integer
- A: an array of numbers, with entries $A[1], A[2], \cdots, A[n]$

Postcondition, Q_1:

- A sorted array A' with $A'[1] \leq A'[2], \leq \cdots \leq A'[n]$

Example:

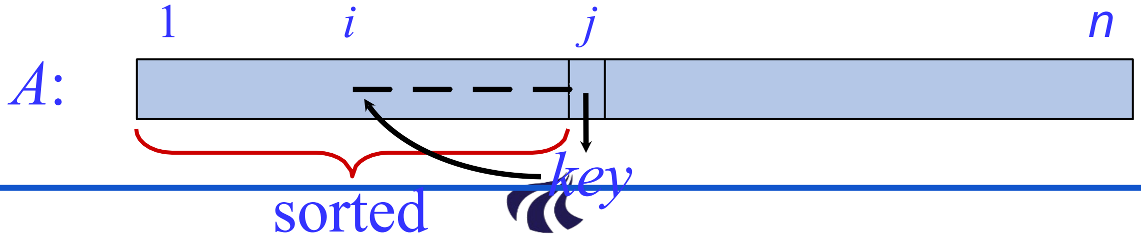
- Precondition/input: $A = [8\ 5\ 1\ 2\ 4\ 3]$
- Postcondition/output: $A = [1\ 2\ 3\ 4\ 5\ 8]$

Insertion sort

Idea: Remove an element from an unsorted input array and insert in the correct position in an already-sorted, but partial list that contains elements from the input list.

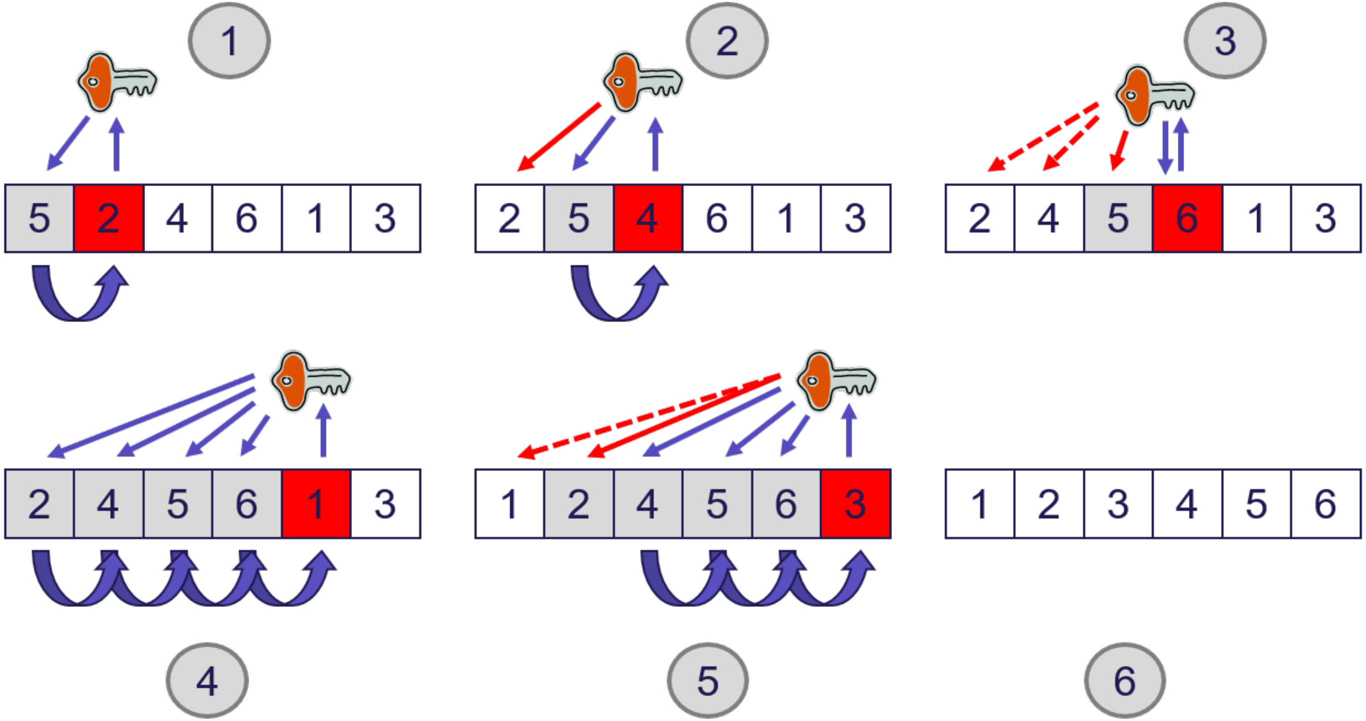
pseudocode

INSERTION-SORT (A, n)
  for $j \leftarrow 2$ **to** n
    do $key \leftarrow A[j]$
      $i \leftarrow j - 1$
      while $i > 0$ and $A[i] > key$
        do $A[i+1] \leftarrow A[i]$
          $i \leftarrow i - 1$
        $A[i+1] = key$



Insertion Sort: Example in Picture

```
INSERTION-SORT (A, n)
  for j ← 2 to n
      do key ← A[j]
         i ← j – 1
         while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i – 1
             A[i + 1] = key
```



Running Time Analysis

- How long does an algorithm take to execute as a function of the input size?
  - On a particular input, it is the **number of primitive operations** (steps) executed.

Running time analysis:

- The goal is to define steps to be **machine-independent**
- Each line of pseudocode(algorithm) requires a constant amount of time.
  - Each execution of line i takes the same amount of time, c_i . Line j may take a different amount of time c_j than line 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.
- Analysis can be **worst-case** (usually), **average-case**, or **best case**

Insertion sort: Running time analysis

- The **time complexity,** $\boldsymbol{T(n)}$, of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size, n.

	cost	times
for j := 2 **to** *n* **do**	c1	n
key := A[j]	c2	n-1
// Insert A[j] into A[1..j-1]		
i := j-1	c3	n-1
while i>0 and A[i]>key **do**	c4	$\sum_{j=2}^{n} t_j$
A[i+1] := A[i]	c5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:= key	c7	n-1

Analysis of Insertion Sort

- Assume that the i th line takes time c_i , which is a constant.
- For $j = 2, 3, \ldots, n$,
  - 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:

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

- For insertion sort:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

  - The running time depends on t_j which varies according to the input.

Running Time Analysis

- Worst-case:
  - T(n) = maximum running time of an algorithm on any input of size n.
  - Useful for determining if algorithms used in real-time systems with deadlines will possibly fail
- Average-case:
  - T(n) = expected running time of an algorithm over all inputs of size n.
  - Need assumption of the statistical distribution of inputs.
  - Algorithms typically perform this way
  - Usually closer to worst case than to best case
- Best-case:
  - T(n) = minimum running time of an algorithm on any input of size n.
  - For comparison sometimes it is easiest to compare algorithms how they work in best cases.

Running Time Analysis

- What is the worst-case running time for insertion?
  - It depends on the speed of our computer:
    - relative speed (on the same machine),
    - absolute speed (on different machines).

- **Key Idea:**
  - Ignore **machine-dependent constants**.
  - Look at growth of $T(n)$ as $n \rightarrow \infty$ ‘
    - “Asymptotic Analysis”

Analysis of Insertion Sort

- **Best case:** The array is already sorted.
  - Always find that $A[i] <= key$ the first time the while loop test is run (when $i = j - 1$) => $t_i = 1$
  - Running time:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7\,(n-1)$$

  - Best case running time is a linear function of n: $T(n) = an + b$

- **Worst case**
  - The array is in reverse sorted order.
  - Always find that $A[i] > key$ in while loop test.
  - Must compare key with all elements to the left of the jth position => compare with $j - 1$ elements => $t_j = j$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n-1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\,(n-1)$$

  - Worst case running time is a quadratic function of n: $T(n) = an^2 + bn + c$

Order of Growth/Asymptotic Analysis

- The Θ-notation:

$$\Theta(g(n)) = \{ f(n) : \text{ there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that } 0 \le c1g(n) \le f(n) \le c2g(n) \text{ for all } n \ge n0 \}$$

- Idea:
  - Drop low-order terms and ignore leading constants
  - Examples:
    - $an^3 + bn^2 + cn + d = \Theta(n^3)$
    - $an^2 + d = \Theta(n)$

- What is the worst-case and best-case complexity of insertion sort?

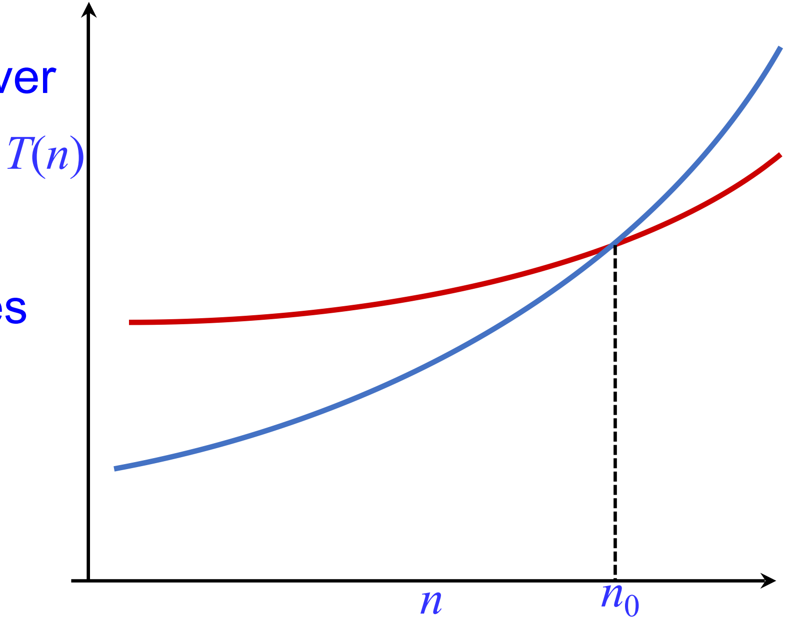
Asymptotic Performance

- When n gets large enough:
  - A $\Theta(n^2)$ algorithm is always better than $\Theta(n^3)$ algorithm.

- Asymptotically slower algorithms must however not be ignored

- Real-world design situations often call for a careful balancing of engineering objectives

- Asymptotic analysis is a useful tool to help to structure our thinking.



Correctness of Algorithms

Partial Correctness

Partial correctness:

- If
  - inputs satisfy the precondition, P, and
  - algorithm, S is executed

- then either
  - S halts and its inputs and outputs satisfy the postcondition, Q

- or
- S does not halt, at all.

This is generally written as

$$\{P\}\ S\ \{Q\}$$

Proof Correctness

To prove correctness, we associate a number of **assertions** (statements about the state of the execution) **with specific checkpoints** in the algorithm.

Consider an algorithm S:

- **Divide S into sections $S_1; S_2; \cdots ; S_N$**
  - assignment statements
  - loops
  - control statements
  - (other programming constructs)
- **Identify intermediate assertions R_i so that**
  - $\{P\}\ S_1\ \{R_1\}$
  - $\{R_1\}\ S_2\ \{R_2\}$
  - …..
  - $\{R_{N-1}\}\ S_N\ \{R_N\}$

After proving each of these assertions, we can then conclude that

$$\{P\}\ S_1;\ S_2; \cdots ;\ S_N\ \{Q\},$$

and

$$\{P\}\ S\ \{Q\}$$

Example: Proof of Partial Correctness

Problem: Finding the largest entry in an integer array

Precondition, *P*:

- n: a positive integer
- A: an integer array with length n and entries $A[0], \cdots, A[n-1]$

Postcondition, *Q*:

- Output is the integer m such that
  - $0 \le m < n$, $A[m] \ge A[k]$ for every k ($0 \le k < n$)

Constraints:

- Inputs (and other variables) have not changed

Algorithm FindMax

```
intFindMax(A, n)
m = 0
k = 1
while k < n do
    if A[k] > A[m] then
        m = k;
    end
    k = k+1;
end
```

Example: Proof of Partial Correctness

```
Algorithm FindMax
intFindMax(A, n)
m = 0
k = 1
while k < n do
    if A[k] > A[m] then
        m = k;
    end
    k = k+1;
end
```

Proof of each Section:

Prove the correctness of each section of the algorithm using the intermediate assertion I:

- First Section: P $m = 0; k = 1$ R
  - correctness is trivial
- Second Section:
  R while... end while Q
  - proof is required

Note: In this course, we will focus on proving correctness of simple loops and recursive algorithms

Correctness of Loops

- **Problem**:

  {P} while G do S end while {Q}

- **Observation:**
  - There are generally some conditions that we expect to hold at the beginning of every execution of the body of a loop. Such a condition is called **a loop invariant**.

- To prove correctness of a loop, we need **to identify Loop Invariant R** and prove:
  - **Base Property (Initialization):** P implies that R is True before the first iteration of the loop
  - **Inductive Property (Maintenance):** If R is True before an iteration and the loop guard G is True, then R is True after the iteration
    - Note: This is essentially a proof by mathematical induction that the loop invariant holds after zero or more executions of the loop body.
  - Prove the correctness of the postcondition (Termination)
    - if the loop terminates after zero or more iterations, the Truth of R implies that Q is satisfied.

Proof of the loop invariant in find maximum algorithm

Claim: Algorithm max correctly finds the maximum of the input list of integers

Proof:

- **Base property (Initialization):**
  - At the first iteration: $j = 2:\ \max = a_1 \text{ if } a_1 > a_2 \text{ or } \max = a_2\, if\, a_2 > a_1$
  - The maximum of 2 elements in the set is returned
  - Conclusion: the base ppty holds
- **Inductive property (maintenance):**
  - For each element, a_i the algorithm compares with the current maximum and update the maximum only if a_i is larger
  - Each execution of the loop maintains the property
- **Termination:**
  - Whats happen when the for loop reaches $i > n$
  - With $i = n + 1$, we have checked and compare all elements with the maximum
    - This indicate that the variable maximum holds the largest integer in the list

Conclusion: The max algorithm is correct (works as desired)

Correctness of Insertion Sort

- How do we proof the correctness of the insertion sort algorithm?

INSERTION-SORT (A, n)

for $j \leftarrow 1$ **to** $n-1$
  do $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i \geq 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] = key$

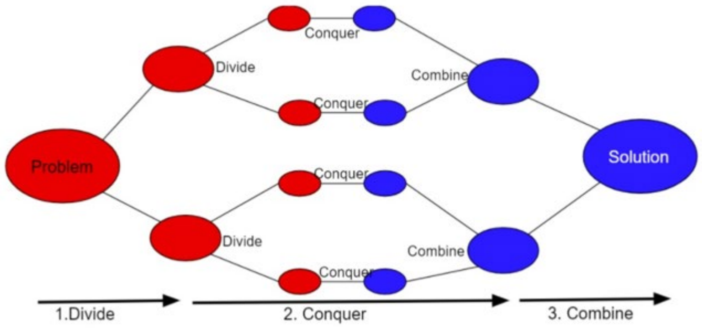
Design of Algorithms

AALBORG UNIVERSITET

Divide and Conquer

- Principle:
  - If the problem size is small enough to solve it trivially, solve it.

  Else:
  - **Divide**: Decompose the problem into one or more disjoint subproblems.
  - **Conquer**: Use divide and conquer recursively to solve the subproblems.
  - **Combine**: Take the solutions to the subproblems and combine the solutions into a solution for the original problem.



Divide and Conquer – Merge-sort example

MERGE-SORT $A[1 \,.\,.\, n]$

If $n = 1$, done.

Recursively sort $A[\,1 \ldots \lceil n/2 \rceil]$ and $A[\,\left\lceil \frac{n}{2} \right\rceil + 1 \ldots n\,]$.

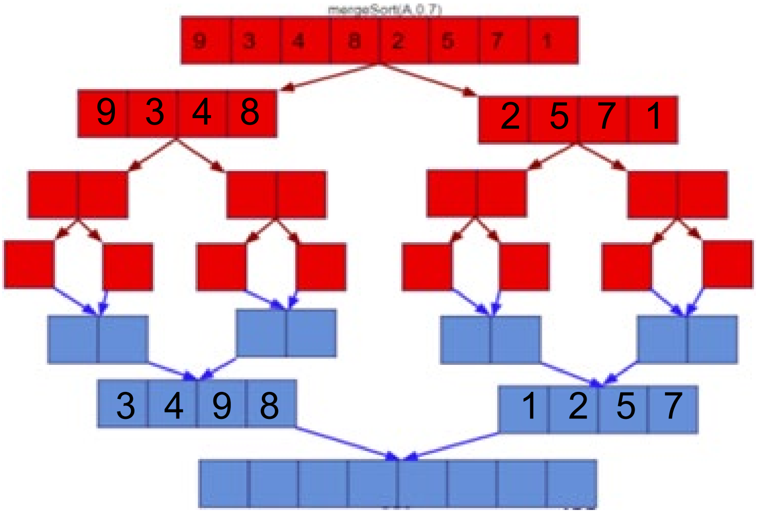
"***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**

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

Merge-sort example:

- We illustrate the divide and conquer principle via illustration of the mergesort algorithm on the unsorted array, A = [9, 3, 4, 8, 2, 5, 7, 1].




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

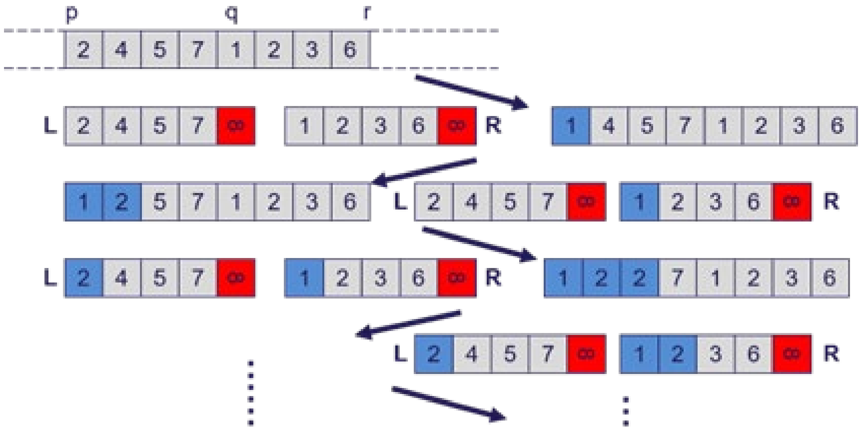
The Merge Algorithm

Input: Array A and indices p; q; r such that

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

Output: The two subarrays are merged into a single sorted subarray in $A[p \dots r]$

- We can implement the merge algorithm to run in linear time.



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
```

Analysis of Divide and Conquer Algorithms

- We use a **recurrence** equation to describe the running time of a divide-and-conquer algorithm.
- If the problem size is small enough 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 original size: in merge sort, $a = b = 2$.
  - Let the time to divide a size-n problem be $D(n)$
  - Each subproblem takes $T(n/b)$ time to solve => we spend $aT(n/b)$ time solving subproblems.
  - Let the time to combine solutions be $C(n)$
- We get the recurrence:

$$T(n) = \begin{cases} \Theta(1), & if\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise \end{cases}$$

Analyzing merge sort

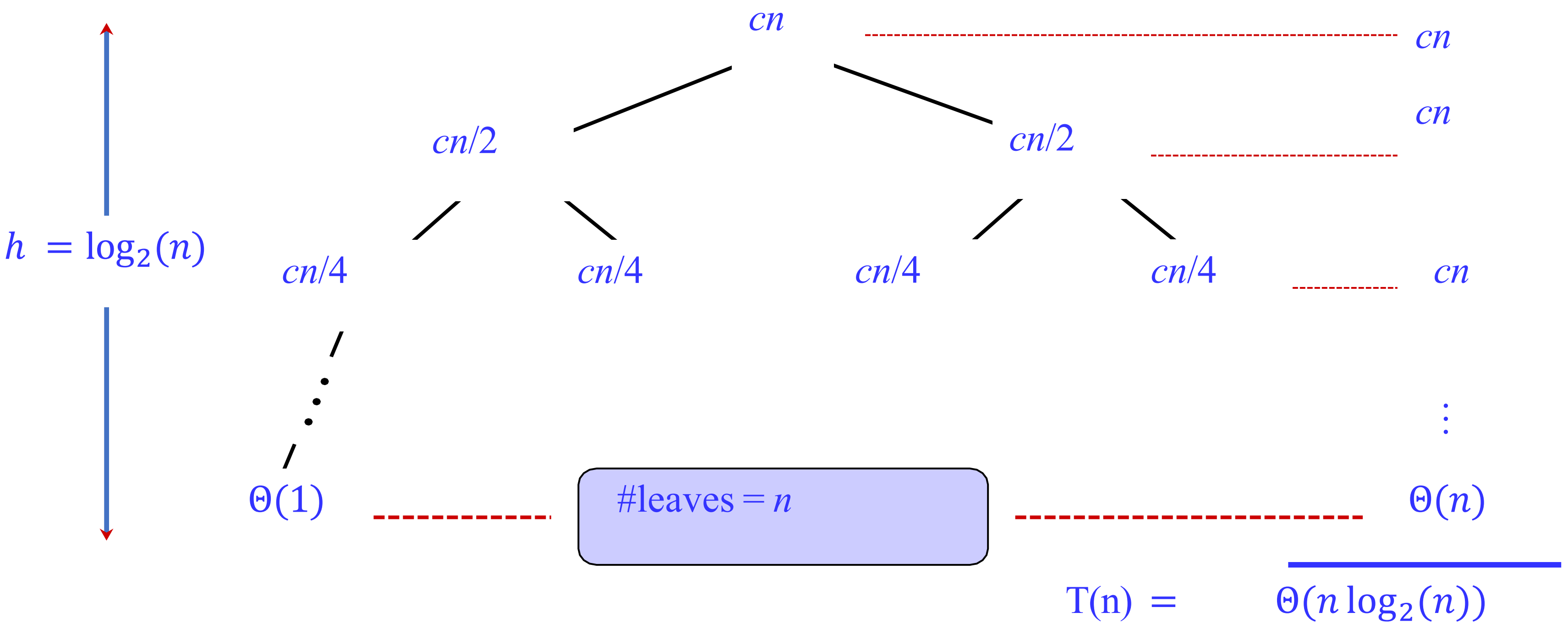
- For simplicity, assume that n is a power of 2 => each divide step yields two subproblems, both of size exactly n/2.
- The base case occurs when n=1.
- When $n \geq 2$, time for merge sort steps:
  - Divide: Just compute q as the average of p and r => $D(n) = \Theta(1)$
- Conquer: Recursively solve 2 subproblems, each of size $n/2$ => $2T(n/2)$.
- Combine: MERGE on an n-element subarray takes $C(n) = \Theta(n)$ time
- Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, $D(n) + C(n) = \Theta(n)$
- Recurrence for merge-sort is then:

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & if\ n \geq 1 \end{cases}$$

Solving recurrences

Recursion Tree

- Solve $2T\left(\frac{n}{2}\right) + cn$, where c > 0 is a constant.



Repeated substitution

- We now apply repeated substitution to find the running time of merge sort (assuming n = 2k)

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & if\ n \geq 1 \end{cases}$$

- Repeated substitution:

$$\begin{aligned} T(n) &= 2\boldsymbol{T}(\boldsymbol{n}/\mathbf{2}) + n \\ &= \mathbf{2}(\mathbf{2}\boldsymbol{T}(\boldsymbol{n}/\mathbf{4}) + \boldsymbol{n}/\mathbf{2}) + n \\ &= 2^2(\boldsymbol{T}(\boldsymbol{n}/\mathbf{4})) + 2n \\ &= 2^2(\mathbf{2}\boldsymbol{T}(\boldsymbol{n}/\mathbf{8}) + \boldsymbol{n}/\mathbf{4}) + 2n \\ &= 2^3T(n/8) + 3n \end{aligned}$$

- Observe the pattern in the above equations:

$$\begin{aligned} T(n) &= 2^k\, T(n/2^k) + kn \\ &= 2^{\log(n)}\, T\left(\frac{n}{n}\right) + n\log_2(n)\, T(n) \\ &= n + n\log_2(n) \end{aligned}$$

Summary and Key Take-aways

- $\Theta(n \log_2(n))$ grows more slowly than $\Theta(n^2)$.
  - merge sort is asymptotically better than insertion sort in the worst case.
  - In practice, merge sort beats insertion sort for n > 30 or so.

- To prove the correctness of algorithms, we identify a loop invariant and prove its properties:
  - **Initialization**
  - **Maintenance**
  - **Termination**

- To analyze the running time of divide and conquer algorithms:
  - We write the recurrence equation and solve it using
    - Masters theorem
    - Recursion tree
    - Repeated substitution

- Next lecture: Growth of functions and asymptotic analysis