Data Structures & Algorithms (F28DA)

Design Techniques – Backtracking, Divide & Conquer

Kathrin Stark

Heriot-Watt University, 2024

Today You'll

- Learn how **backtracking** helps in improving brute force algorithm by pruning the search space
- Learn about the general pattern of **backtracking** algorithms
- Be reminded of the general pattern of Divide-and-Conquer Algorithms
- Learn three ways on how to reason about the asymptomatic runtime of Divide-and-Conquer algorithms

Backtracking

Brute Force

Brute Force Algorithm = A straightforward and exhaustive problem-solving technique that

1. Enumerates all possible solutions
2. Checks whether one of these is actually a solution

without leveraging optimisation or heuristic strategies.

Correctness? – Trivial as all solutions are considered

Efficiency? – Often quite inefficient

Runtime:

- **Number of possible solutions:** g(n)
- **Time to check a solution:** f (n)
- **Total time**: g(n) * f(n)

Problem: Possibly a huge problem space
Idea: Can we **prune** the problem space by looking at **partial solutions** already?

Example: Subset Sum Problem

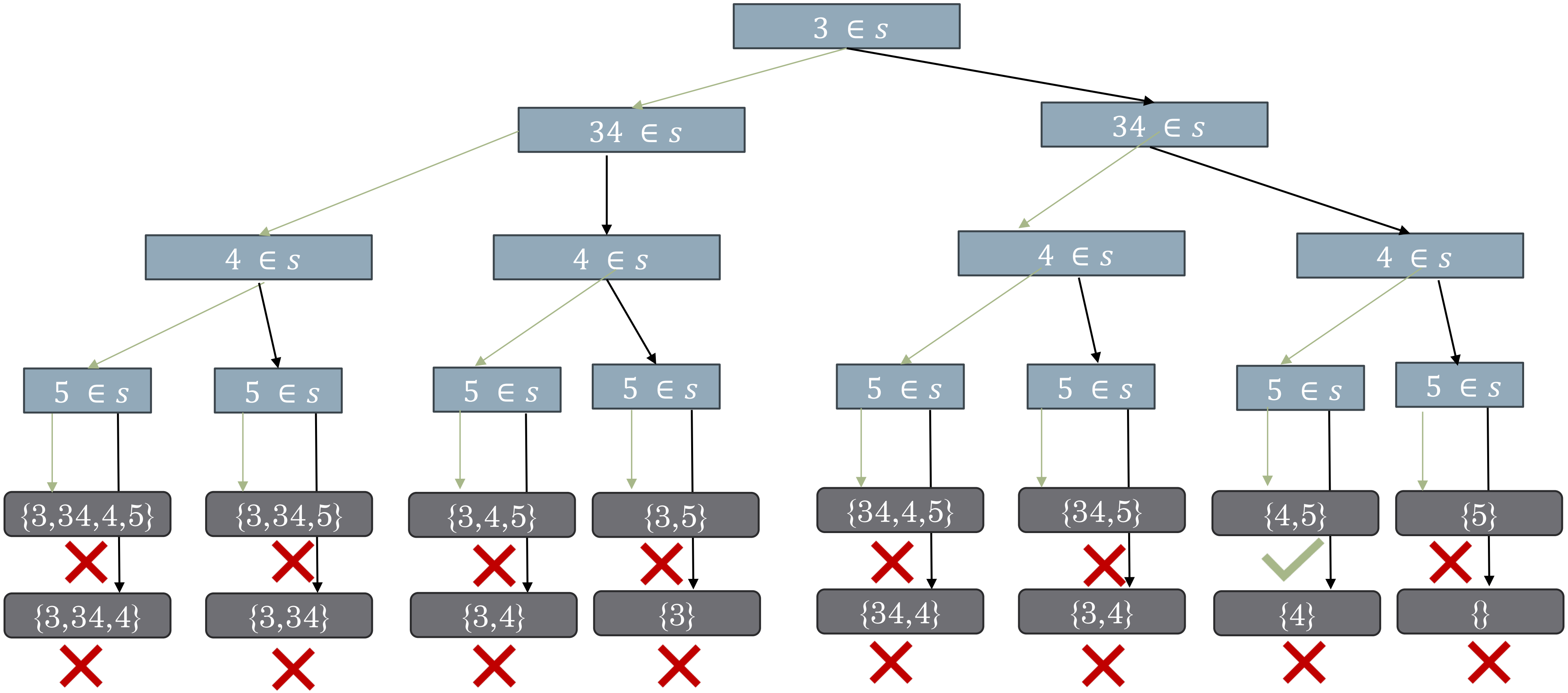
Subset Sum Problem:
Input: A set S of n positive integers, an integer k
Output: True if there is a subset s of integers in S that sum to k

Example. Let S = $\{3, 34, 4, 5\}$. $k = 9$.

Example: Subset Sum Problem
The Complete Search Tree

Brute Force:
1. Enumerates all possible solutions
2. Checks whether one of these is actually a solution

Let S = *{3, 34, 4, 5}*. *k* = *9*.

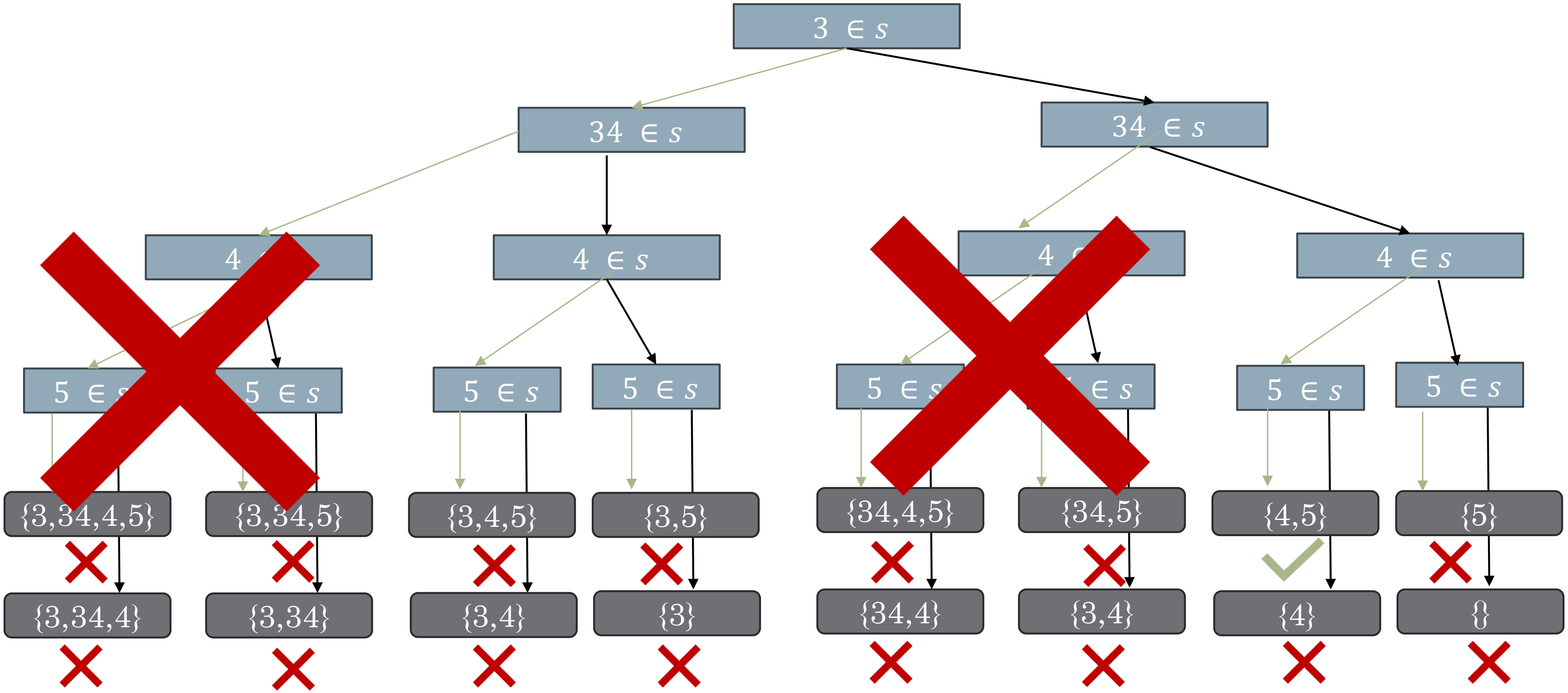


Example: Subset Sum Problem

The Pruned Search Tree

Observation:
We can prune parts of the search tree whenever the sum on the way to our set is already > k.

Let S = $\{3, 34, 4, 5\}$. $k = 9$.



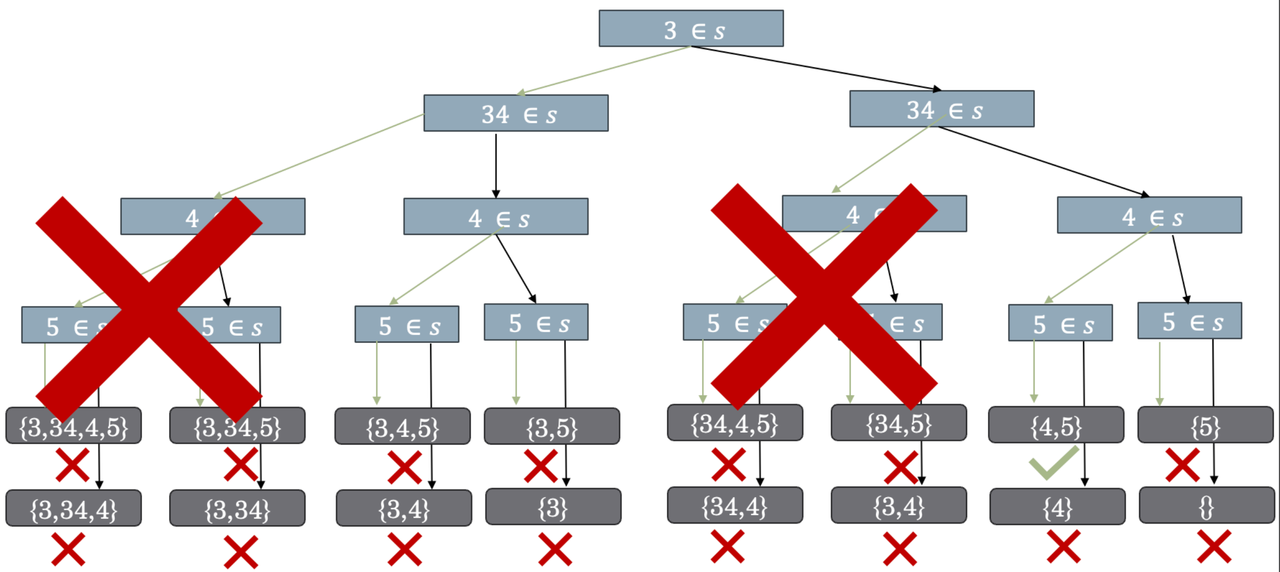
General Observation

… we can put this in a **general template**.
What we need:
- A way to build up solutions incrementally.
- A way to reject partial solutions.

Example: Subset Sum Problem
The Pruned Search Tree

Observation:
We can prune parts of the search tree whenever the sum on the way to our set is already > k.

Let S = {3, 34, 4, 5}. k = 9.



Backtracking

Backtracking Algorithm = An exhaustive problem-solving technique that traverses through possible search paths to locate solutions or dead ends.

1. Start with a subproblem configuration.
2. Choose the most promising extending subproblem configuration.
3. If the configuration should get to a dead end, get back to the last decision point. If a solution is found, stop.

Correctness? – Yes, if all viable paths are considered

General Backtracking Template

Algorithm Backtrack(x):
  Input: A problem instance x for a hard problem
  Output: A solution for x or "no solution" if none exists

```
F ← {(x, ∅)}.          {F is the "frontier" set of subproblem configurations}
while F ≠ ∅ do
  select from F the most "promising" configuration (x, y)
  expand (x, y) by making a small set of additional choices
  let (x_1, y_1), (x_2, y_2), ..., (x_k, y_k) be the set of new configurations.
  for each new configuration (x_i, y_i) do
    perform a simple consistency check on (x_i, y_i)
    if the check returns "solution found" then
      return the solution derived from (x_i, y_i)
    if the check returns "dead end" then
      discard the configuration (x_i, y_i)        {Backtrack}
    else
      F ← F ∪ {(x_i, y_i)}        {(x_i, y_i) starts a promising search path}
return "no solution"
```

Source: Algorithm Design by Goodrich & Tamassia, p. 627

Example: Subset Sum Problem
Pseudocode

```
Algorithm findSubset(S, k)
Input: A set S of n positive integers, an integer k
Output: True if there is a subset s of integers in S that sum to k
\\ we found a solution
if (k == 0) then
 return True
\\ there are no more candidates
if (S.isEmpty()) then
 return False

s <- S.get() // next candidate integer

if (s > k) then
   \\ we're overshooting k - so don't include s
  return findSubset (S\{s}, k)
else
  \\ look both in the case that s is included and not
  \\ || is short-circuiting
 findSubset (S\{s}, k - s) || findSubset (S\{s}, k)
```

Initial subconfiguration:
The full set S, k.

Extending subconfiguration:
Removing an element s from S, k or k - s

Dead end: If no more elements are in S/ k gets negative

Solution: k is 0

Example: Subset Sum Problem

Pseudocode

```
Algorithm findSubset(S, k)
Input: A set S of n positive integers, an integer k
Output: True if there is a subset s of integers in S that sum to k
\\ we found a solution
if (k == 0) then
 return True
\\ there are no more candidates
if (S.isEmpty()) then
 return False

s <- S.get() // next candidate integer

if (s > k) then
   \\ we're overshooting k - so don't include s
  return findSubset (S\{s}, k)
else
  \\ look both in the case that s is included and not
  \\ || is short-circuiting
 findSubset (S\{s}, k - s) || findSubset (S\{s}, k)
```

What's the complexity?

Where Backtracking Is Used

Efficiency: Not necessarily a better asymptotic complexity but can reduce runtime drastically in practice (e.g., SAT solving).

The **heuristic chosen** is important.

Examples:

- **Subset Sum Problem**
- **N-Queens problem**: Given an NxN chessboard, place N queens such that no two queens threaten each other.
- Solving a partially completed **Sudoku**
- **Knight's Tour:** Find a sequence of moves for a knight on a chessboard such that the knight visits every square exactly once
- (Later): Find a **Hamiltonian circle** in a graph; find a **graph coloring**
- **SAT** solvers

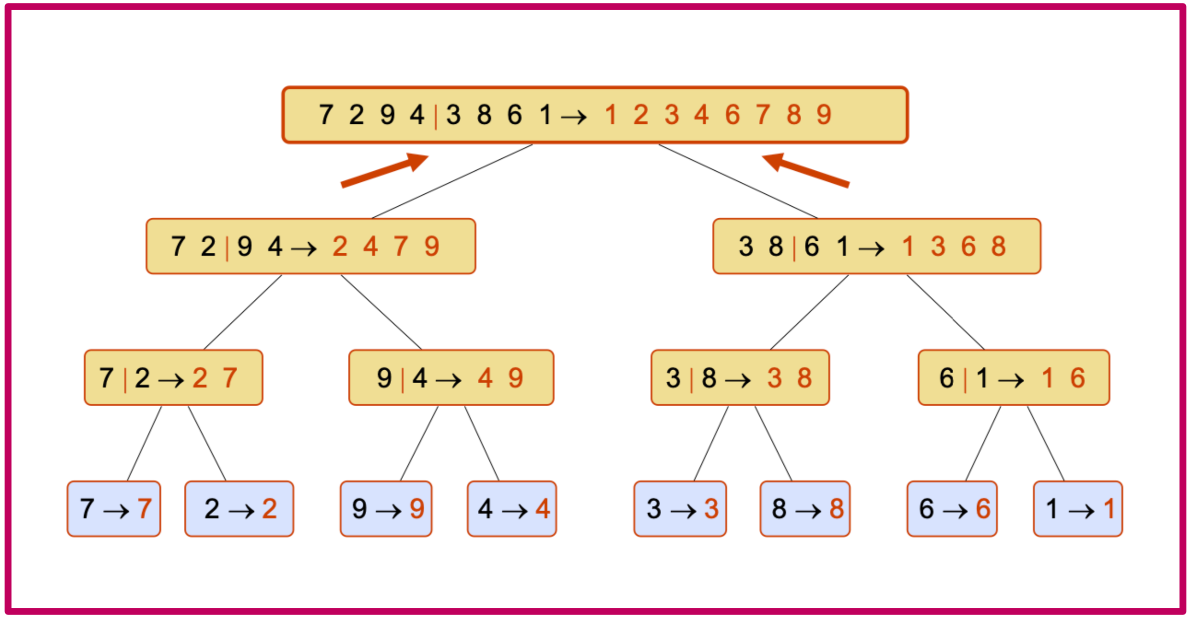
Divide and Conquer

Divide and Conquer Pattern
General Idea

The **Divide-and-Conquer** strategy solves a problem by

1. Breaking it into **subproblems** (= smaller instances of the original problem)
2. Recursively solving the **subproblems** (base case: the problem is small enough to be solved directly)
3. Combining the solutions to get a solution to the original problem.

Reminder: Merge Sort



Idea: Split the list into two and sort every list independently.
Combine the two sorted list with a procedure merge.

Source: F28GS

Breaking it into subproblems
= breaking it into smaller lists

Solving the subproblems
= running the sorting algorithm on the smaller lists

Combining the solutions
= running the merge algorithm

Divide and Conquer Pattern
Runtime

The **Divide-and-Conquer** strategy solves a problem by

1. Breaking it into **subproblems** (= smaller instances of the original problem)
2. Recursively solving the **subproblems** (base case: the problem is small enough to be solved directly)
3. Combining the solutions to get a solution to the original problem.

We often use a **recurrence** to express the running time of a divide and conquer algorithm.

- Time b in the case that n is small, say n <= k
- Divide the problem into a subproblems, each 1/b the size of the original
- Time to divide a size-n problem: D(n)
- Time to combine solutions: C(n)

$$T(n) = \begin{cases} b, & n \leq k \\ a\,T\left(\frac{n}{b}\right) + D(n) + C(n), & n > k \end{cases}$$

Runtime Analysis Merge

Algorithm merge(S_1,S_2,S):

Input: Sequences S_1 and S_2 sorted in nondecreasing order, and an empty sequence S

Output: Sequence S containing the elements from S_1 and S_2 sorted in nondecreasing order, with sequences S_1 and S_2 becoming empty

```
while (not (S1.isEmpty() or S2.isEmpty) do
  if S1.first().element() ≤ S2.first().element() then
    { move the first element of S1 at the end of S }
    S.insertLast(S1.remove(S1.first()))
  else
    { move the first element of S2 at the end of S }
    S.insertLast(S2.remove(S2.first()))
{ move the remaining elements of S1 to S }
while (not S1.isEmpty()) do
  S.insertLast(S1.remove(S1.first()))
{ move the remaining elements of S2 to S }
while (not S2.isEmpty()) do
  S.insertLast(S2.remove(S2.first()))
```

Runtime:

Source: Algorithm Design by Goodrich & Tamassia

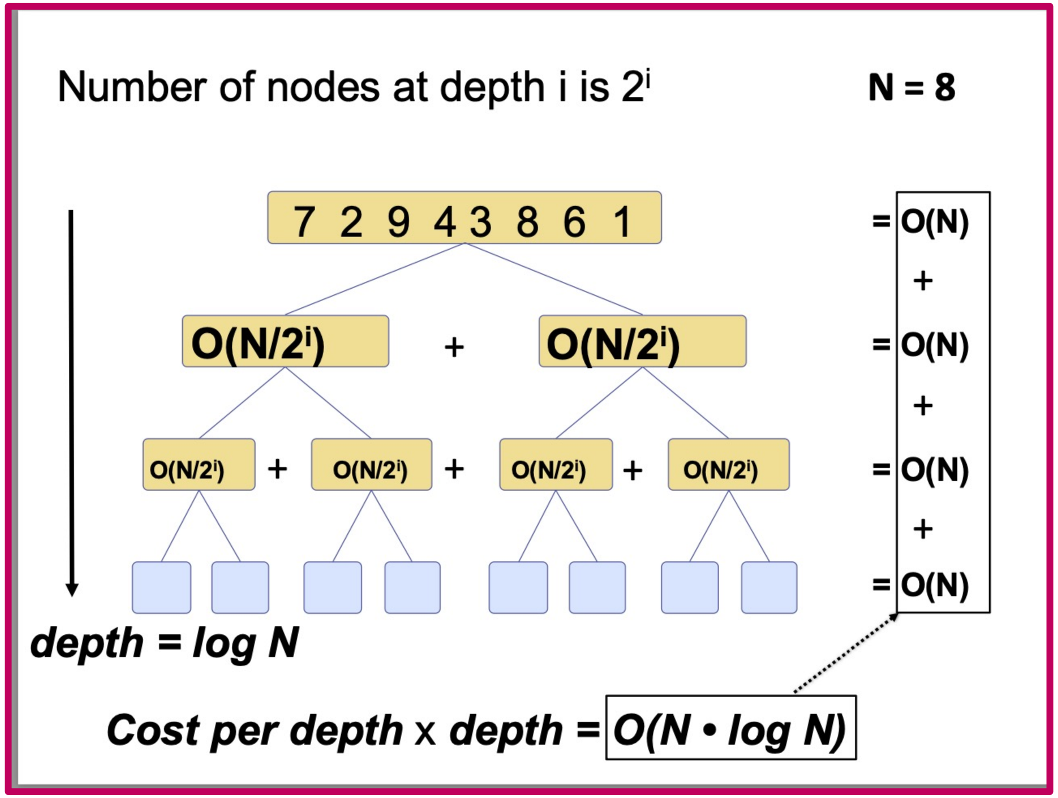
Runtime Analysis Merge Sort

```
Algorithm mergesort (S)
Input: Sequence S
Output: Sequence S in sorted order
if S.size() < 2 then
  return S
// Split S into two equally long sequences (± 1)
(S1, S2) <- split S
S1 <- mergesort (S1)
S2 <- mergesort (S2)
S <- emptySequence
return merge (S1, S2, S)
```

Recurrence equations for runtime T(n):

$$T(n) = \begin{cases} b, & n < 2 \\ 2\,T\left(\frac{n}{2}\right) + c\,n, & n \geq 2 \end{cases}$$

Possibility 1: Argue on the Recursion Tree



Problem:
Works here but is not as easy when the recursion pattern gets more complicated.

Possibility 2:
Guess a solution and prove it correct

- ☐ Guess an expression for the solution. The expression can contain constants that will be determined later.
- ☐ Use induction to find the constants and show that the solution works.

Let us apply this method to MERGE-SORT.

The recurrence of MERGE-SORT implies that there exist two constants $c, d > 0$ such that

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

Guess. There is some constant $a > 0$ such that $T(n) \leq an \lg n$ for all $n > 2$ that are powers of 2.

Source: https://www.cs.ox.ac.uk/files/12487/divide-and-conquer.pdf

Possibility 2:
Guess a solution and prove it correct

Test. For $n = 2^k$, by induction on k.

Base case: $k = 1$

$$T(2) = 2c + 2d \leq a\,2\lg 2 \qquad \text{if } a \geq c + d$$

Inductive step: assume $T(n) \leq an\log n$ for $n = 2^k$.
Then, for $n' = 2^{k+1}$ we have:

$$\begin{aligned} T(n') &\leq 2a\tfrac{n'}{2}\lg\left(\tfrac{n'}{2}\right) + d\,n' \\ &= an'\lg n' - an'\lg 2 + d\,n' \\ &\leq an'\lg n' \qquad \text{if } a \geq d \end{aligned}$$

In summary: choosing $a \geq c + d$ ensures $T(n) \leq an\lg n$,
and thus $T(n) = O(n\log n)$.
A similar argument can be used to show that $T(n) = \Omega(n\log n)$.
Hence, $T(n) = \Theta(n\log n)$.

Source: https://www.cs.ox.ac.uk/files/12487/divide-and-conquer.pdf

Possibility 3: The Master Theorem
A "Recipe" for Recurrence Equations in Divide & Conquer

Let

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d, \end{cases}$$

for $d \geq 1, a > 0, c > 0$ and $b > 1$ and f(n) be a function that is positive for $n \geq d$.

Theorem 5.6 [The Master Theorem]: *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\varepsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are small constants $\varepsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.*

Source: Algorithm Design Goodrich & Tamassia, p. 268

Example Master Theorem

Merge Sort

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

- What's a and b? What's $\log_b a$?
- Of what complexity is f(n)?
- Does any of the cases apply?

Let

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d, \end{cases}$$

for $d \geq 1, a > 0, c > 0$ and $b > 1$ and f(n) be a function that is positive for $n \geq d$.

Theorem 5.6 [The Master Theorem]: *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\varepsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are small constants $\varepsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.*

Example Master Theorem

```
Algorithm findMax (a, l, u)
Input: Array a of size u – l = n >=1
Output: Max. element in a between l
and u -1

if l+1 = u then
    return a[l]
mid = ⌊(l+u)/2⌋
return max (findMax (a, l, mid),
findMax (a, mid, u))
```

$$T(n) = \begin{cases} c, & n < 2 \\ 2\,T\left(\frac{n}{2}\right) + d, & n \geq 2 \end{cases}$$

- What's a and b? What's $\log_b a$?
- Of what complexity is f(n)?
- Does any of the cases apply?

Let

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d, \end{cases}$$

for $d \geq 1, a > 0, c > 0$ and $b > 1$ and f(n) be a function that is positive for $n \geq d$.

Theorem 5.6 [The Master Theorem]: *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\varepsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are small constants $\varepsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.*

You Learned

- How **backtracking** helps in improving brute force algorithm by pruning the search space
- The general pattern of **backtracking** algorithms
- About the general pattern of Divide-and-Conquer Algorithms
- Three ways on how to reason about the asymptomatic runtime of Divide-and-Conquer algorithms

Reading

- Chapter 1.4 of Algorithms by Sedgewick & Wayne
- Chapter 5.2 of Algorithm Design Goodrich & Tamassia
- Chapter 13.5 of Algorithm Design Goodrich & Tamassia

Optional Reading

- Geogebra.org - allows you to plot graphs
- Big-O Complexity Sheet (https://www.bigocheatsheet.com) – Overview of complexity for most standard algorithms
- Base.cs podcast on O-notation