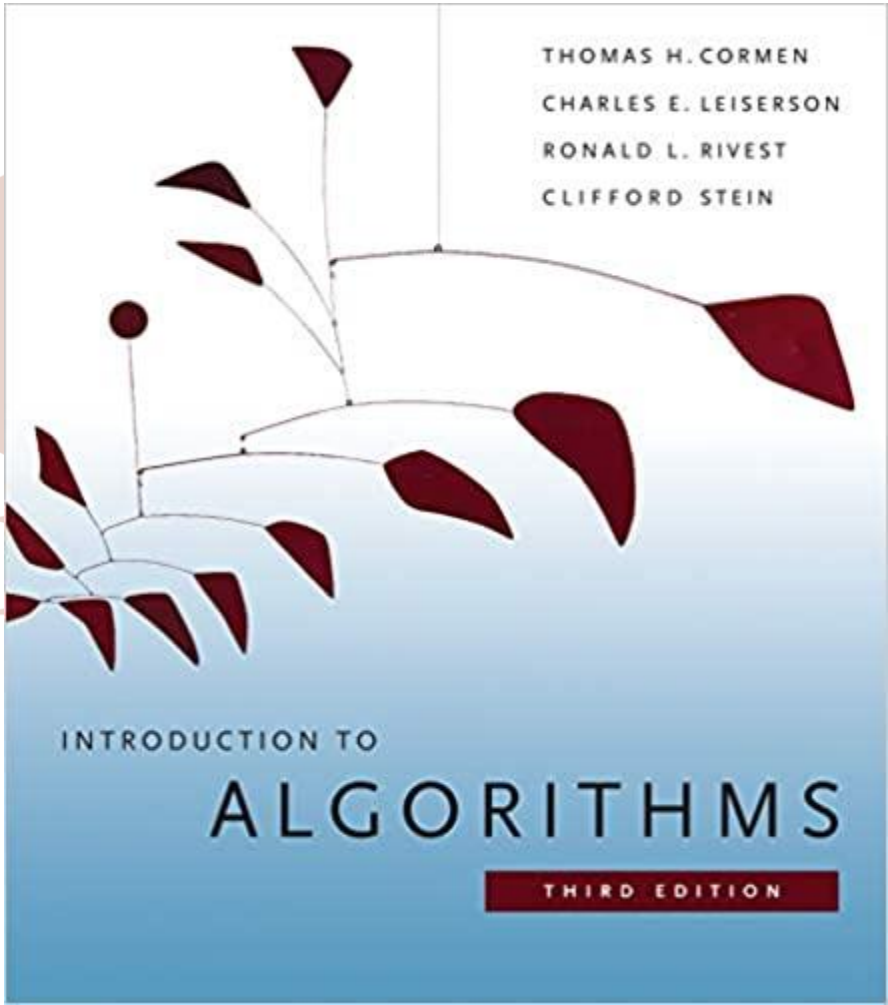
CS 07540 Advanced Design and Analysis of Algorithms

Week 6

- Exhaustive Search
- Decrease-and-Conquer
- Divide-and-Conquer
- Transform-and-Conquer
- Backtracking



Brute Force and Exhaustive Search

Brute-force search or **exhaustive search** is a problem-solving technique which consists of systematically checking all possible solution candidates for whether they satisfy the desired properties.

Examples of exhaustive search algorithms are:

- Selection Sort
- Bubble Sort
- Traveling Salesman Problem
- Knapsack Problem
- Depth-First Search
- Breadth-First Search
- Many, many more…

We will look at Depth-first search (**DFS**) and **Breadth-First Search (BFS)** in the context of graph algorithms later. For now, consider the Assignment Problem as an example of exhaustive search. This will highlight that pruning techniques are necessary to make exhaustive search computationally feasible.

The Assignment Problem

There are n people who need to be assigned to execute n jobs, one person per job. The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i,j]$ for each pair $i,j = 1,2,\dots,n$. The problem is to find an assignment with the minimum total cost.

Since the number of permutations to be considered for the general case of the assignment problem is $n!$ exhaustive search is impractical for all but very small instances of the problem. There is a much more efficient algorithm for this problem called the Hungarian method after the Hungarian mathematicians König and Egerváry in $O(n^4)$, which later was improved to $O(n^3)$, that means this problem can be solved in polynomial time. A variation of this problem is the Stable Marriage Problem.

Example: Find an optimal solution to this assignment problem by hand.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

What tools do we need for a brute force/exhaustive search solution?

Decrease-and-Conquer

Decrease-and-conquer is related to the **divide-and-conquer** we will look at next, but they are different in complexity.

Decrease-and-conquer involves reducing the problem into **one** smaller subproblem, while divide-and-conquer reduces the problem into **several** similar subproblems. The main idea will be to

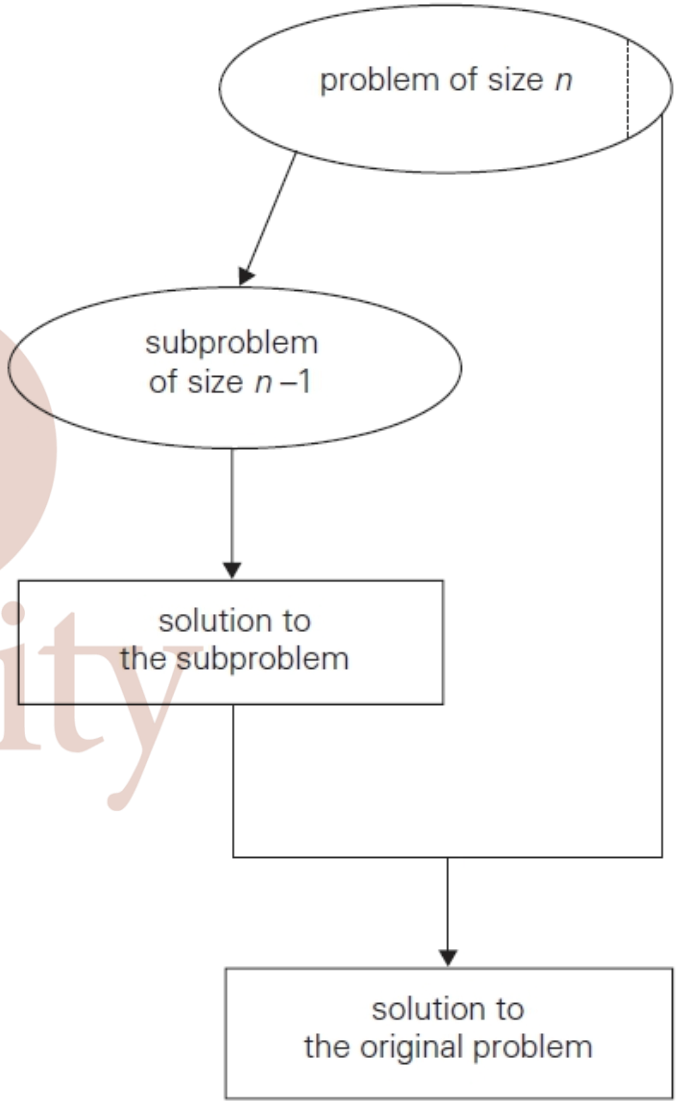
1. **Decrease** / reduce the problem instance to a smaller instance of the same problem.
2. **Conquer** the problem by solving the smaller one first.
3. **Extend** the solution of the smaller instance back to the original problem.

There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

Decrease-by-a-constant

We know some examples of **decrease-by-a-constant** by the name of tail-recursion problems: exponentiation in a loop, such as $a^n = a^{n-1} \cdot a$, or calculating $n!$ as $(n-1)! \cdot n$.

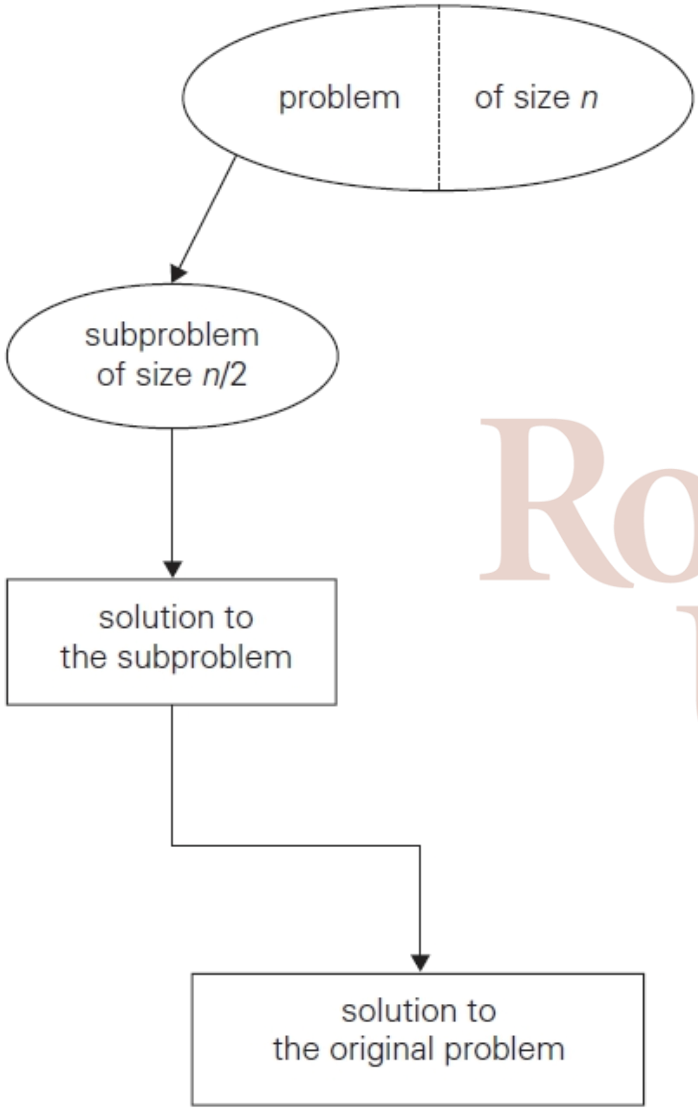


Insertion Sort Again

We already looked at **Insertion Sort**. Here we use it as an example of decreasing a problem of size n to a problem of size $n-1$ repeatedly until we reach a base case. It is based on a recursive idea: Sorting an array with n elements is easy when the leading array with $n-1$ elements is already sorted. We only need to find the correct place for the new element. This can be easily implemented non-recursively in a double-loop. In fact, Jon Bentley has a three-line implementation:

```
for i = [1, n)
    for (j = i; j > 0 && x[j-1] > x[j]; j--)
        swap(j-1, j)
```

Hence, we start with an array of size 1, increasing the size by 1 each iteration, and find the correct place for the new element.



Decrease-by-a-factor

Decrease-by-a-factor reduces a problem to one subproblem of smaller size by a factor. For example, if computing a^n recursively by

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{if } n \text{ is even and positive} \\ \left(a^{(n-1)/2}\right)^2 \cdot a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

leads to a $\Theta(\log(n))$ algorithm. This idea differs from Divide-and-conquer in that it only reduces the problem to one subproblem.

Another example of *Decrease-by-a-factor* is binary search, which we already looked at.

Variable-size-decrease

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

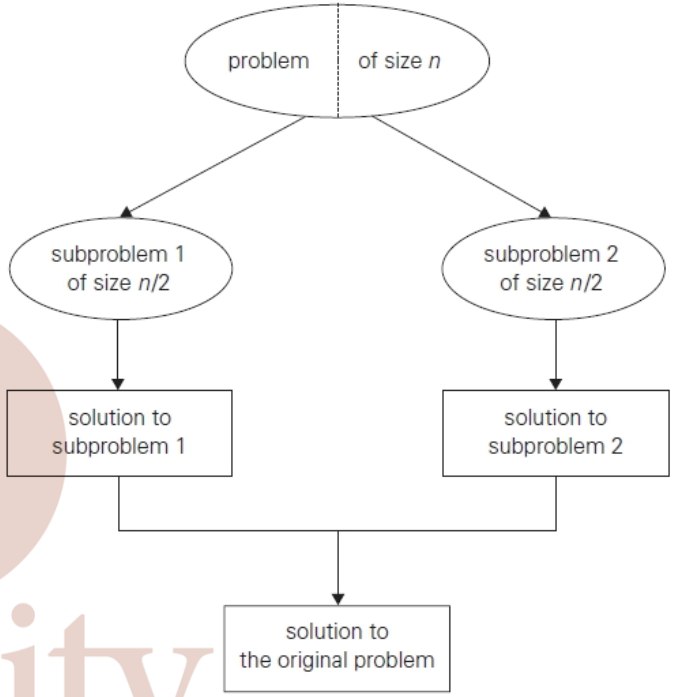
decreases arguments, but neither in a constant nor percentage style.

An example of variable-size-decrease is search in a generic *Binary Search Tree*. The problem of finding a key is reduced to a smaller problem each time we make a comparison, since the left or right subtree that we need to traverse will be smaller. We just don't know by how much.

In contrast, search in a **balanced BST** is an example of **divide-and-conquer**. In a weight-balanced BST, left and right subtrees have about the same number of elements.

Divide-and-Conquer

Divide-and-conquer algorithms work according to the following general plan:

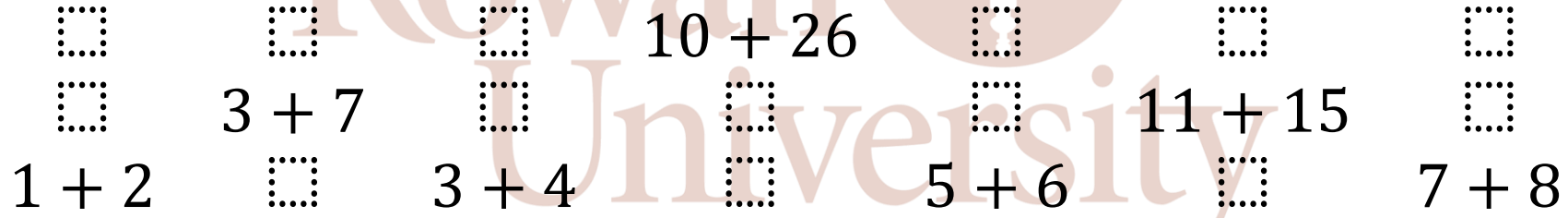


1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Example: Consider summing up eight numbers

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$$

The brute-force algorithm takes seven additions, which is $O(n)$ where n is the count of numbers. How about creating subproblems of size 2?



It still takes seven additions. The additions are actually the overhead here. There will always be an overhead $f(n)$ associated with subdividing a problem and then putting it back together.

Master Theorem

We can apply the Master Method for recurrences of the form

$$T(n) = a \cdot T\left({}^{n}/_{b}\right) + f(n)$$

where b is the number of subproblems and a of them need to be solved. It is a simplified form of generating functions theory, and the bounds are usually determined by the behavior of $f(n)$. This is a hard problem, in general.

Suppose we have the following recurrence for running time $T(n)$:

$$T(n) = a \cdot T\left({}^{n}/_{b}\right) + f(n)$$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

General Version (CLRS)

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Sketch of Proof for Master Method

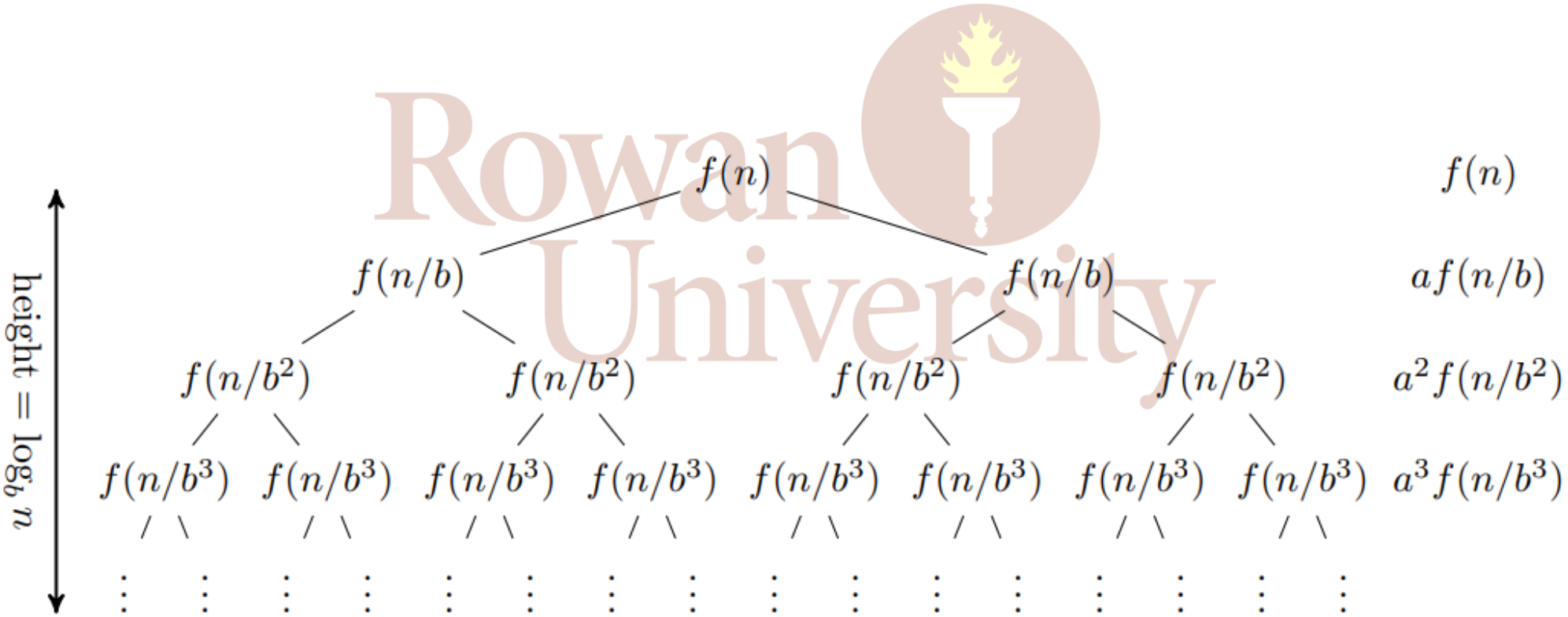
The recurrence

$$T(n) = a \cdot T\left({}^{n}/_{b}\right) + f(n)$$

describes a divide-and-conquer approach which we can visualize with a tree where:

- Each node represents the time $f(n)$ needed as overhead when creating subproblems.
- Each layer represents the subproblems for the next iteration.
- The height of the tree represents the maximum number of subdivisions needed to reach a base case.
- We see that each layer has size a (in the graph $a = 2$).
- The height of the tree is $\log_b n$.

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left({}^{n}/_{b^i}\right) + O\left(n^{\log_b a}\right)$$



Example: Use the Master Theorem on our earlier addition problem to show that the problem is $\Theta(n)$.

Solution: Let the input size $n = 2^k$. The overhead function $f(n) = 1$ (one operation of adding). Then

$$T(n) = 2T\left({}^{n}/_{2}\right) + 1$$

with $a = 2$, $b = 2$, and $d = 0$. Hence $a > b^d$ and the runtime is $\Theta\left(n^{\log_b(a)}\right) = \Theta(n)$.

Applying the Master Theorem

Recall *MergeSort*.

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A) //see below

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array

//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted

//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

$i \leftarrow 0;\ \ j \leftarrow 0;\ \ k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

  if $B[i] \leq C[j]$

    $A[k] \leftarrow B[i];\ i \leftarrow i+1$

  else $A[k] \leftarrow C[j];\ j \leftarrow j+1$

  $k \leftarrow k+1$

if $i = p$

  copy $C[j..q-1]$ to $A[k..p+q-1]$

else copy $B[i..p-1]$ to $A[k..p+q-1]$

Example: Use the Master Theorem to show that *MergeSort* has runtime complexity $\Theta(n \log(n))$.

Recall *QuickSort*.

ALGORITHM *Quicksort*$(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//       indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ← Partition(A[l..r]) //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```


It has worst case $O(n^2)$ comparisons when using a poor choice of pivot, but on average its expected runtime is $O(n \log(n))$.

Example: Use the Master Theorem to show that *QuickSort* has runtime $O(n \log(n))$ when the divide step is balanced.

Example: Use the Master Theorem to show that *QuickSort* has runtime $O(n^2)$ when pivot is the minimum or maximum.

Solution: When the pivot is roughly the median,

$$T(n) = 2 \cdot T\left(\left\lfloor n/2 \right\rfloor\right) + \Theta(n)$$

and we recursively call on two subarrays of size $\frac{n-1}{2}$. The Master Theorem gives us a bound of $O\left(n^{\log_2 2} \log(n)\right) = O(n \log(n))$.

Solution: If the pivot is the minimum or maximum of the current subarray, then we get a linked list recurrence rather than a tree:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

Here $\Theta(n)$ is the partitioning cost since we have to go through the array to see that there is noting to be done. The recognition of the empty array takes $T(0) = \Theta(1)$ and can be folded into $\Theta(n)$. So we basically get

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \big(T(0) + \Theta(i)\big) \\ \square &= \Theta(n^2) \end{aligned}$$

Multiplication of Large Integers

The standard approach for multiplying two n-digit numbers has $O(n^2)$ multiplications (and $O(n)$ additions). A 1960 algorithm by then 23-year old student Anatoly Karatsuba provides a $O\left(n^{\log_2(3)}\right)$ bound using divide-and-conquer techniques:

$$T(n) = 3\,T(n/2) + (c_1 n + c_0)$$

for some constants c_1 and c_0.

Horner's Method

Horner's Method (or Horner's Rule) is an algorithm for polynomial evaluation. Since all values in a computer are calculated through additions and multiplications, functions in computers are essentially polynomials, so fast polynomial evaluation is very important. Horner's Rule writes polynomials in a nested form:

$$a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = a_0 + x\left(a_1 + x\left(a_2 + x(\cdots)\right)\right)$$

This allows the evaluation of a polynomial of degree n with n multiplications and n additions, whereas the standard form uses $\sum_{i=0}^{n} i = \frac{1}{2}n(n-1)$ multiplications and n additions.

Other methods of storing and evaluating polynomials is using the point-value representation of a polynomial (a set of $n + 1$ point-value pairs). We will revisit this representation when working with Fourier Transforms.

Transform and Conquer

The next group of techniques is based on the idea of transforming the problem to an equivalent, yet simpler or more convenient form, solving that, and then translating it back into original form. There are three variations to consider:

- Instance Simplification
  - Transformation to a simpler instance of the same problem.
- Change of Representation
  - Transformation to a different representation.
- Problem Reduction
  - Transformation to a different problem for which an algorithm is already available.

Binary Exponentiation – A Change of Representation

Exponentiating by squaring is a general method for fast computation of large positive integer powers of a number.

ALGORITHM *LeftRightBinaryExponentiation*$(a, b(n))$

//Computes a^n by the left-to-right binary exponentiation algorithm
//Input: A number a and a list $b(n)$ of binary digits $b_I, \ldots, b_0$
// in the binary expansion of a positive integer n
//Output: The value of a^n
$product \leftarrow a$
for $i \leftarrow I - 1$ **downto** 0 **do**
  $product \leftarrow product * product$
  if $b_i = 1$ $product \leftarrow product * a$
return $product$

Example: Calculate a^{13} using the binary exponentiation algorithm. Note that $13 = 1101_2$.

In each iteration of the *for*-loop, we have one multiplication, and one test with possible multiplication.

So an upper bound for the number of operations is $4I$, where I is the number of bits in n. That is $O(\log(n))$.

ALGORITHM *LeftRightBinaryExponentiation*$(a, b(n))$

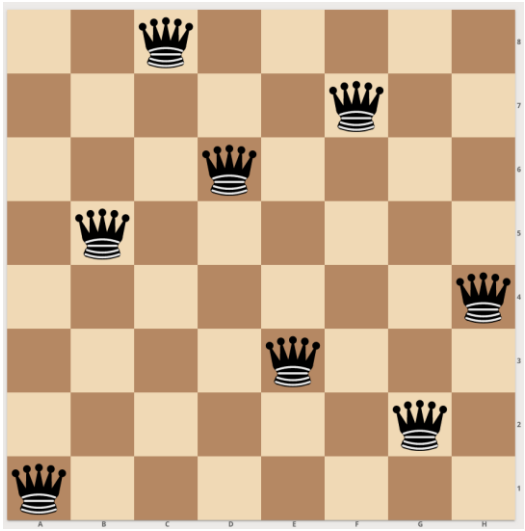
```
//Computes a^n by the left-to-right binary exponentiation algorithm
//Input: A number a and a list b(n) of binary digits b_I, ..., b_0
//         in the binary expansion of a positive integer n
//Output: The value of a^n
product ← a
for i ← I − 1 downto 0 do
    product ← product * product
    if b_i = 1 product ← product * a
return product
```

Backtracking

Backtracking is an algorithmic technique which combines exhaustive search with ideas from decision trees and depth-first search. It constructs partial solutions and evaluates whether such a partial solution can be completed to a solution. If this is not possible, that part of the decision tree is ignored (abandoned) and the algorithm backtracks to a sibling or parent in the decision tree.

n-Queens Problem

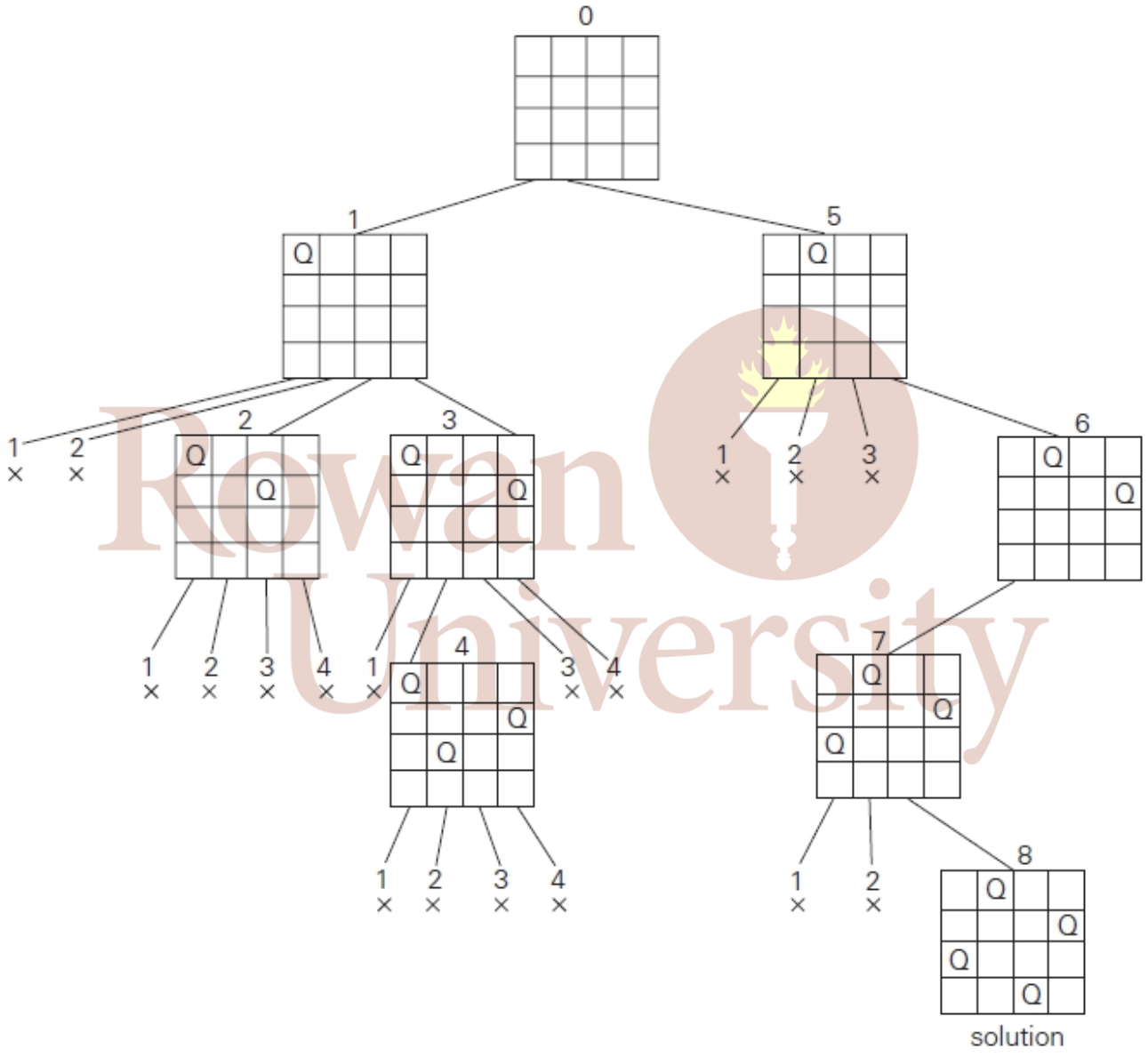
Place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.



ALGORITHM *Backtrack*$(X[1..i])$

//Gives a template of a generic backtracking algorithm
//Input: $X[1..i]$ specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if $X[1..i]$ is a solution **write** $X[1..i]$
else //see Problem 9 in this section's exercises
  for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints **do**
    $X[i+1] \leftarrow x$
    Backtrack$(X[1..i+1])$

Example: Try to systematically construct a solution to the 4-Queens problem. Abandon a track when a partial solution cannot be extended.



Branch-and-Bound

In the standard terminology of optimization problems, a **feasible solution** is a point in the problem's search space that satisfies all the problem's constraints whereas an **optimal solution** is a feasible solution with the best value of the objective function.

Compared to backtracking, branch-and-bound requires two additional items:

- a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
- the value of the best solution seen so far

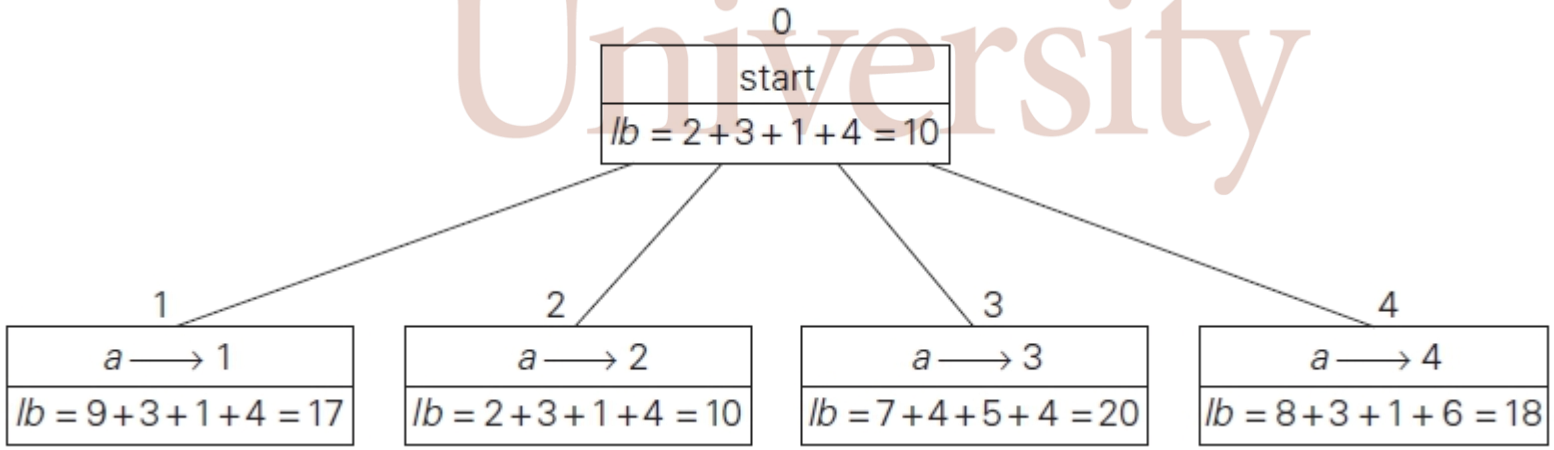
Assignment Problem

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

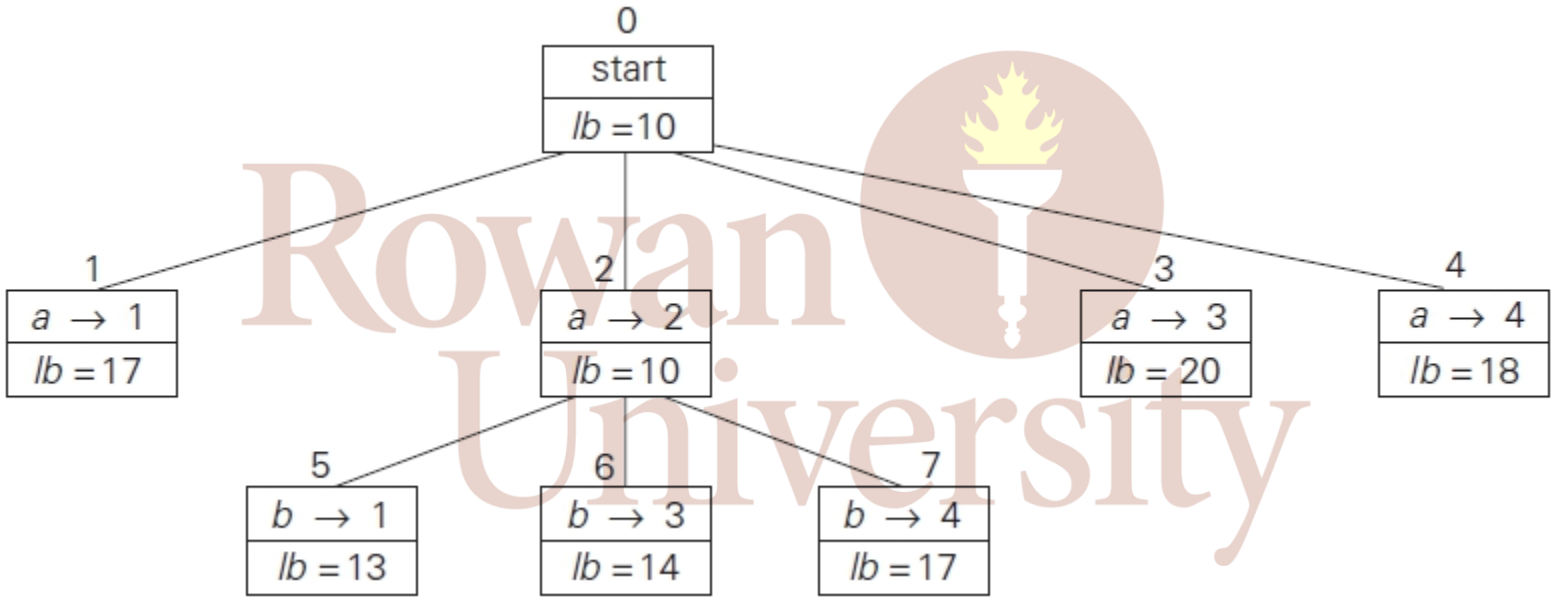
Recall the assignment problem. It is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. That provides a lower bound, even though, in the example given, it does not even correspond to a feasible solution.

Example: Create lower bounds for different assignment choices for person a.

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$



Example: (continued) Continue with the best lower bound to construct a partial solution for b.



Example: (continued) Continue with the best lower bound to construct a partial solution for c (and hence d).