CSE321 Introduction To Algorithm

Nov 10th, 2016
Quick Overview PS

Emre Sercan ASLAN

Outline

- Big Oh Notation
- Master Theorem
- Recurrences
- Brute Force Algorithms
- Exhaustive Search
- Decrease and Conquer Algorithms
- Problems and Solutions

Big Oh Notation

Informally

O	$\approx$	$\leq$
Ω	$\approx$	$\geq$
Θ	$\approx$	$=$
o	$\approx$	$<$
ω	$\approx$	$>$

Formally

***O*-notation**

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}\ .$$

Ω-notation

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\ .$$

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}\ .$$

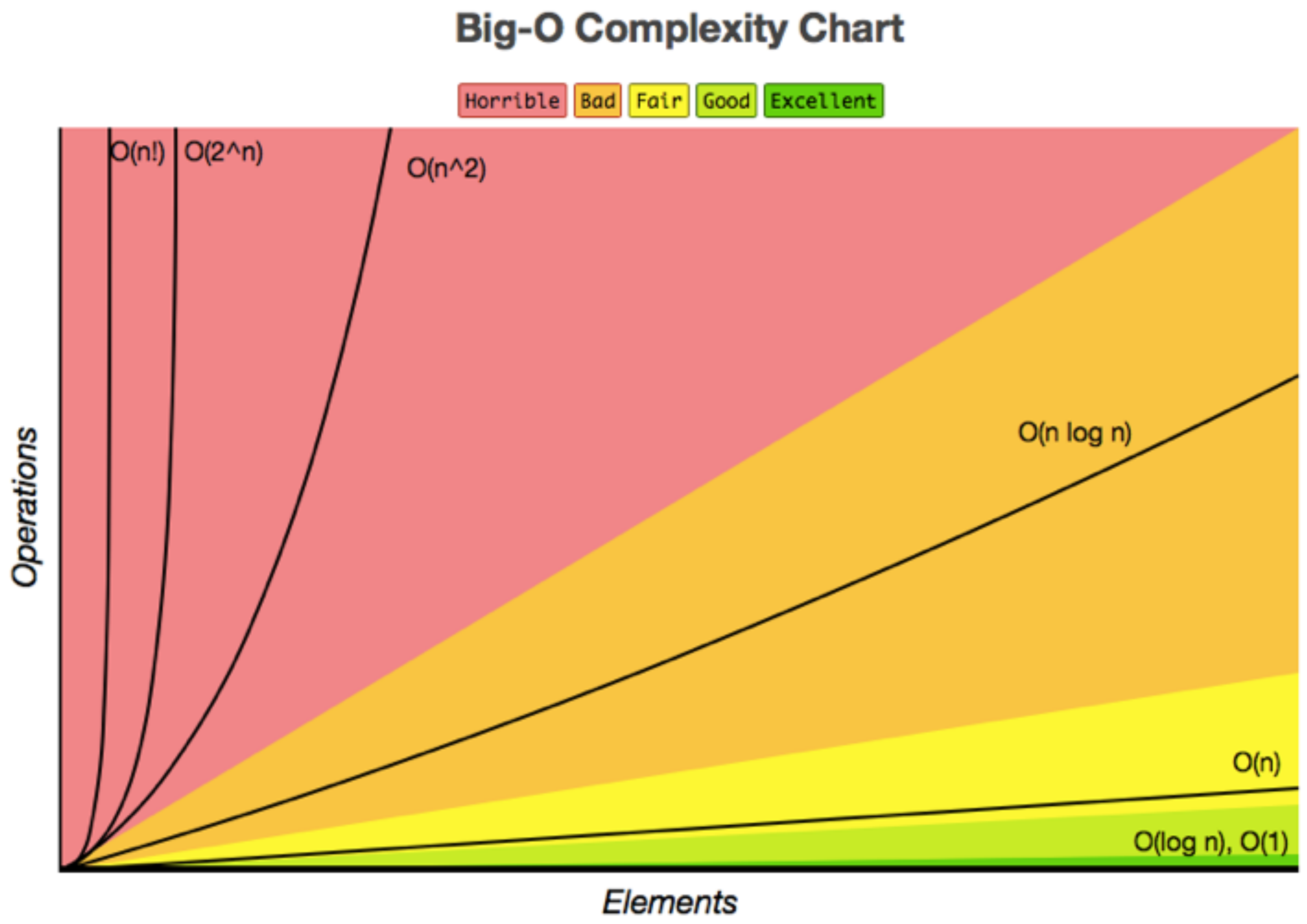
***o*-notation**

$$o(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}\ .$$

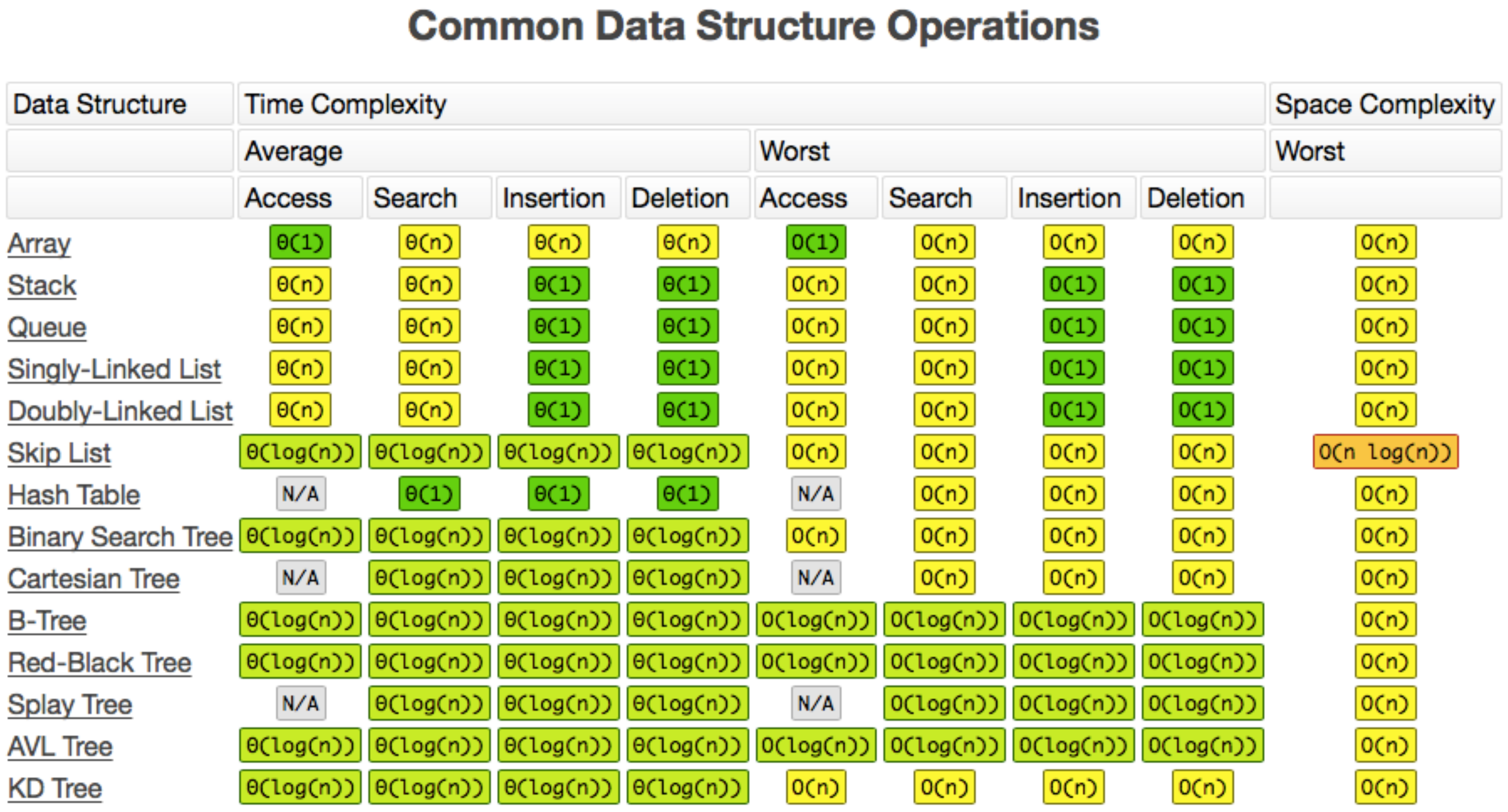
***ω*-notation**

$$\omega(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}\ .$$

Big Oh Notation



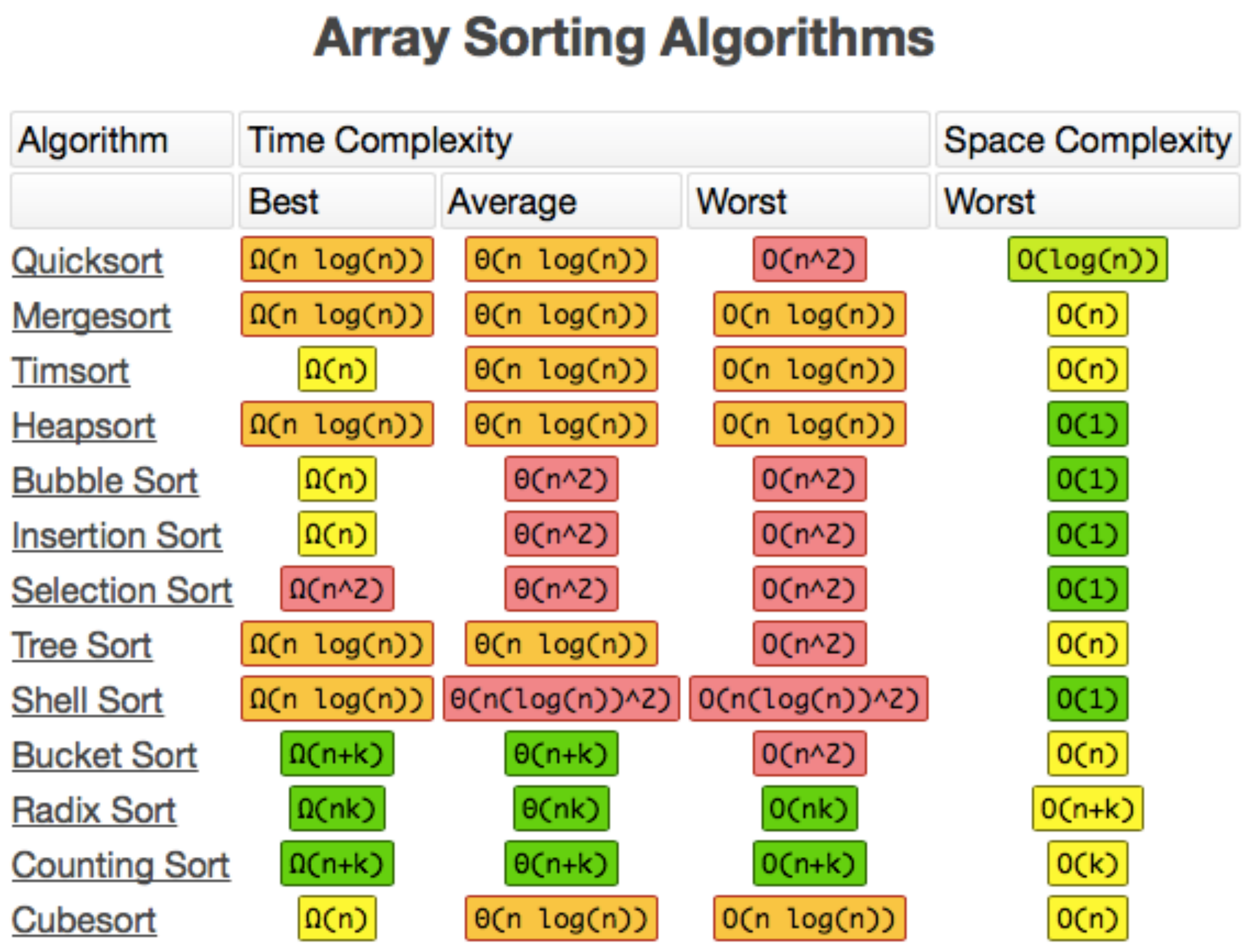
Big Oh Notation



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Big Oh Notation



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

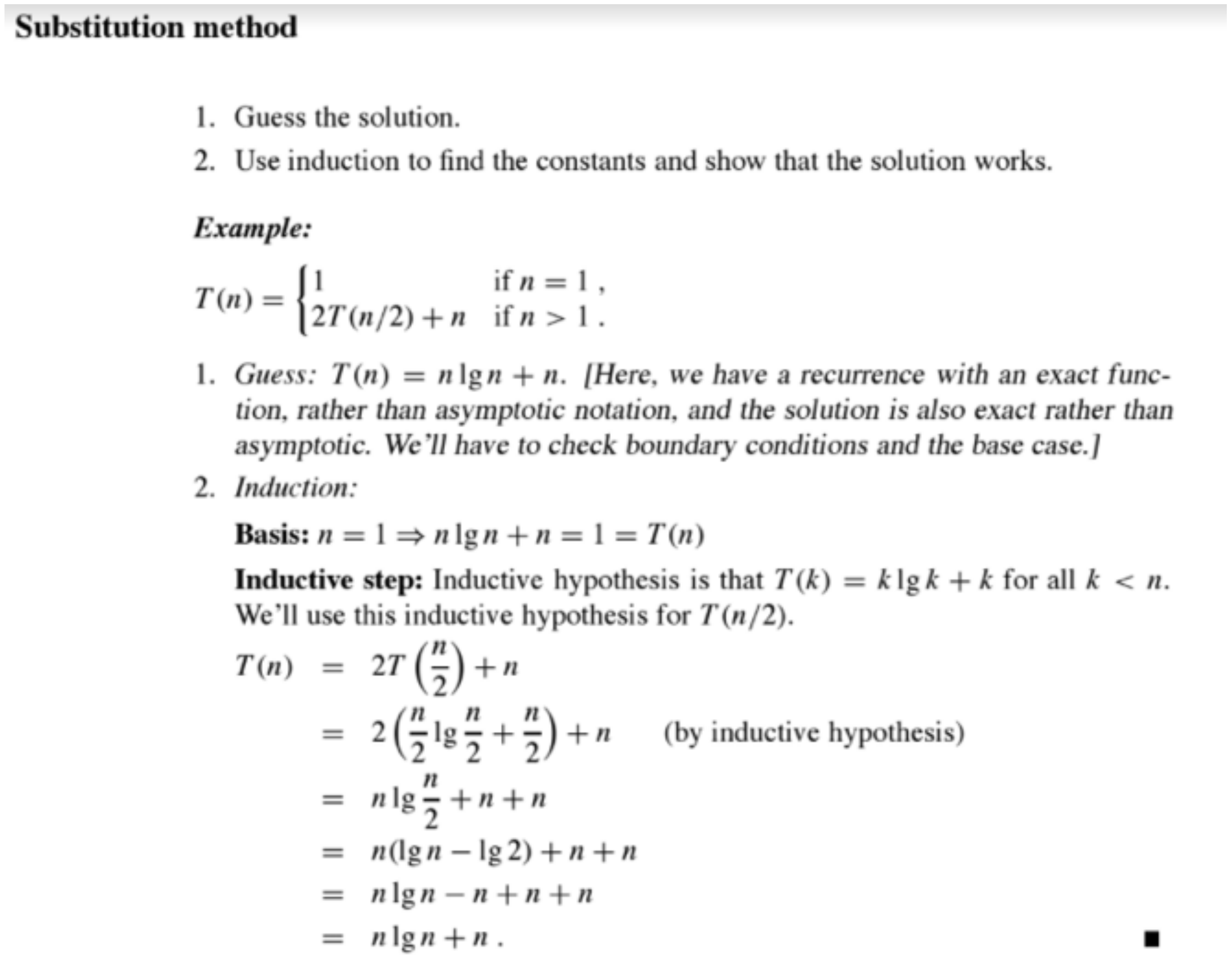
Big Oh Notation Slides Reference

http://bigocheatsheet.com

Recurrences

Substitution Method
Recursion Trees
Master Theorem

Substitution Method



Substitution method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example:

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

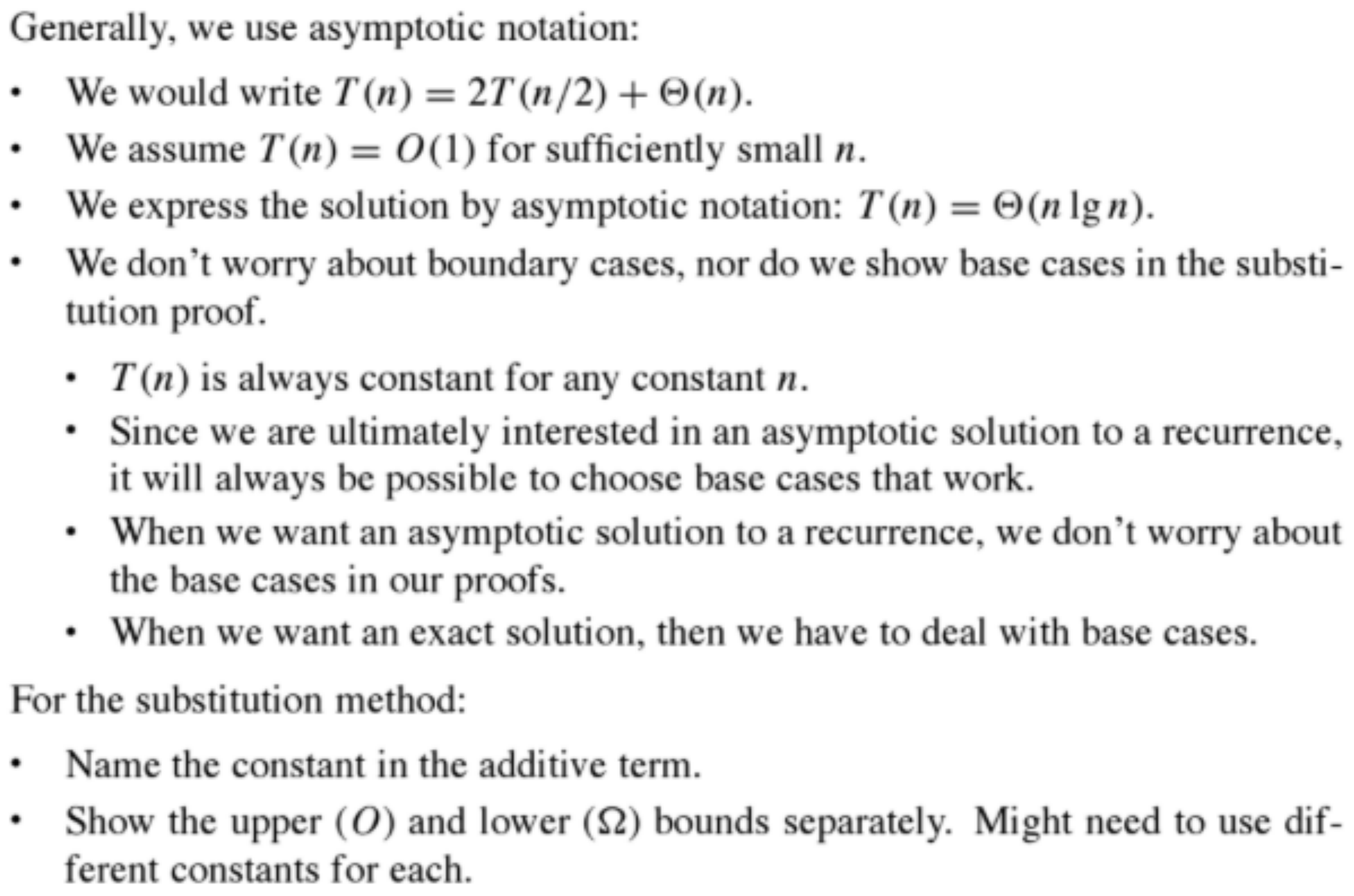
1. *Guess:* $T(n) = n \lg n + n$. *[Here, we have a recurrence with an exact function, rather than asymptotic notation, and the solution is also exact rather than asymptotic. We'll have to check boundary conditions and the base case.]*
2. *Induction:*

   Basis: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

   Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$. We'll use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\ &= n \lg \frac{n}{2} + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n \lg n - n + n + n \\ &= n \lg n + n\,. \qquad \blacksquare \end{aligned}$$

Substitution Method



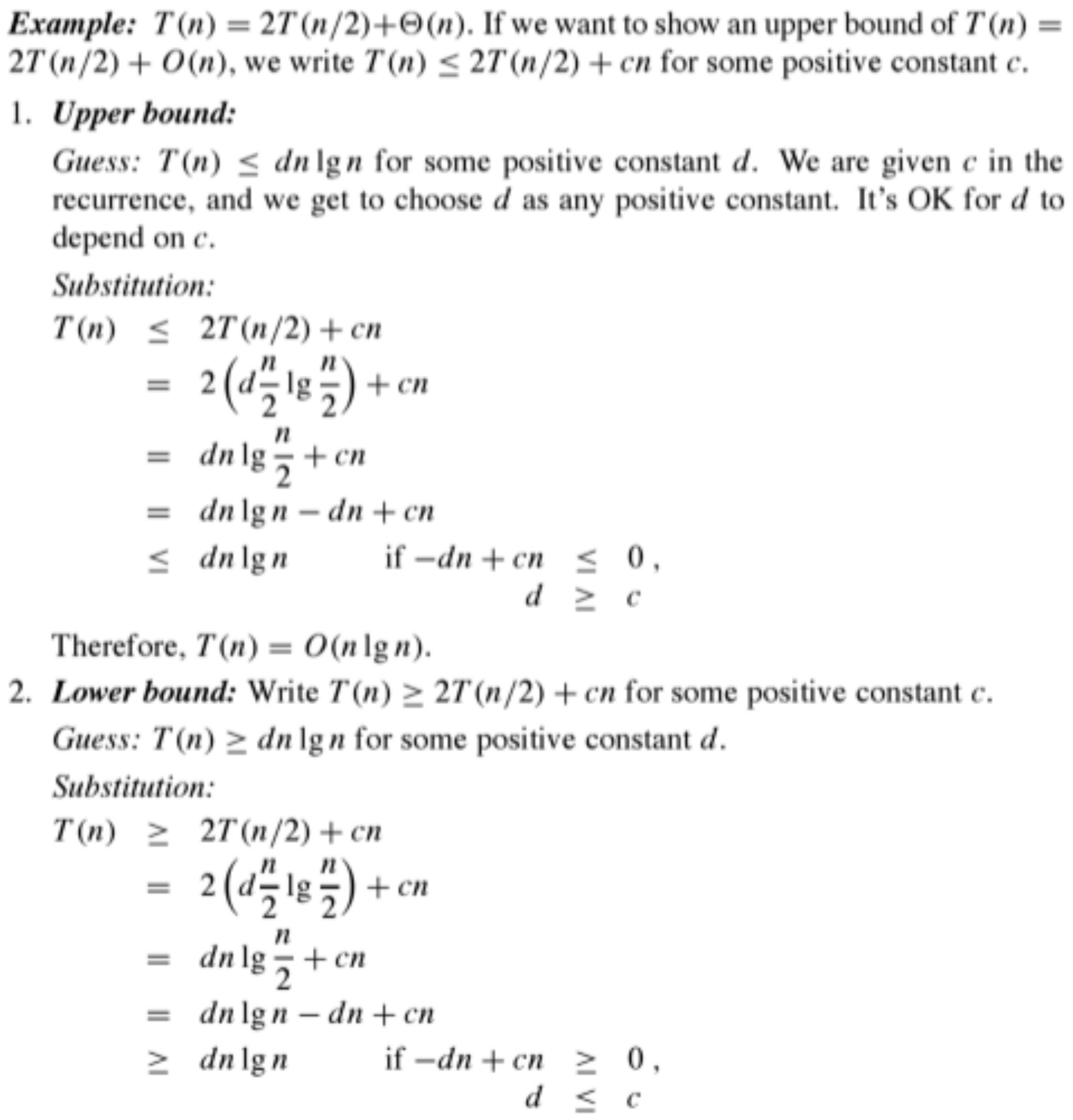
Generally, we use asymptotic notation:

- We would write $T(n) = 2T(n/2) + \Theta(n)$.
- We assume $T(n) = O(1)$ for sufficiently small n.
- We express the solution by asymptotic notation: $T(n) = \Theta(n \lg n)$.
- We don't worry about boundary cases, nor do we show base cases in the substitution proof.
  - $T(n)$ is always constant for any constant n.
  - Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.
  - When we want an asymptotic solution to a recurrence, we don't worry about the base cases in our proofs.
  - When we want an exact solution, then we have to deal with base cases.

For the substitution method:

- Name the constant in the additive term.
- Show the upper (O) and lower (Ω) bounds separately. Might need to use different constants for each.

Substitution Method



Example: $T(n) = 2T(n/2) + \Theta(n)$. If we want to show an upper bound of $T(n) = 2T(n/2) + O(n)$, we write $T(n) \le 2T(n/2) + cn$ for some positive constant c.

1. ***Upper bound:***

   Guess: $T(n) \le dn \lg n$ for some positive constant d. We are given c in the recurrence, and we get to choose d as any positive constant. It's OK for d to depend on c.

   Substitution:

$$\begin{aligned} T(n) &\le 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\le dn\lg n \qquad \text{if } -dn + cn \le 0\,, \\ &\qquad\qquad\qquad\quad d \ge c \end{aligned}$$

   Therefore, $T(n) = O(n\lg n)$.

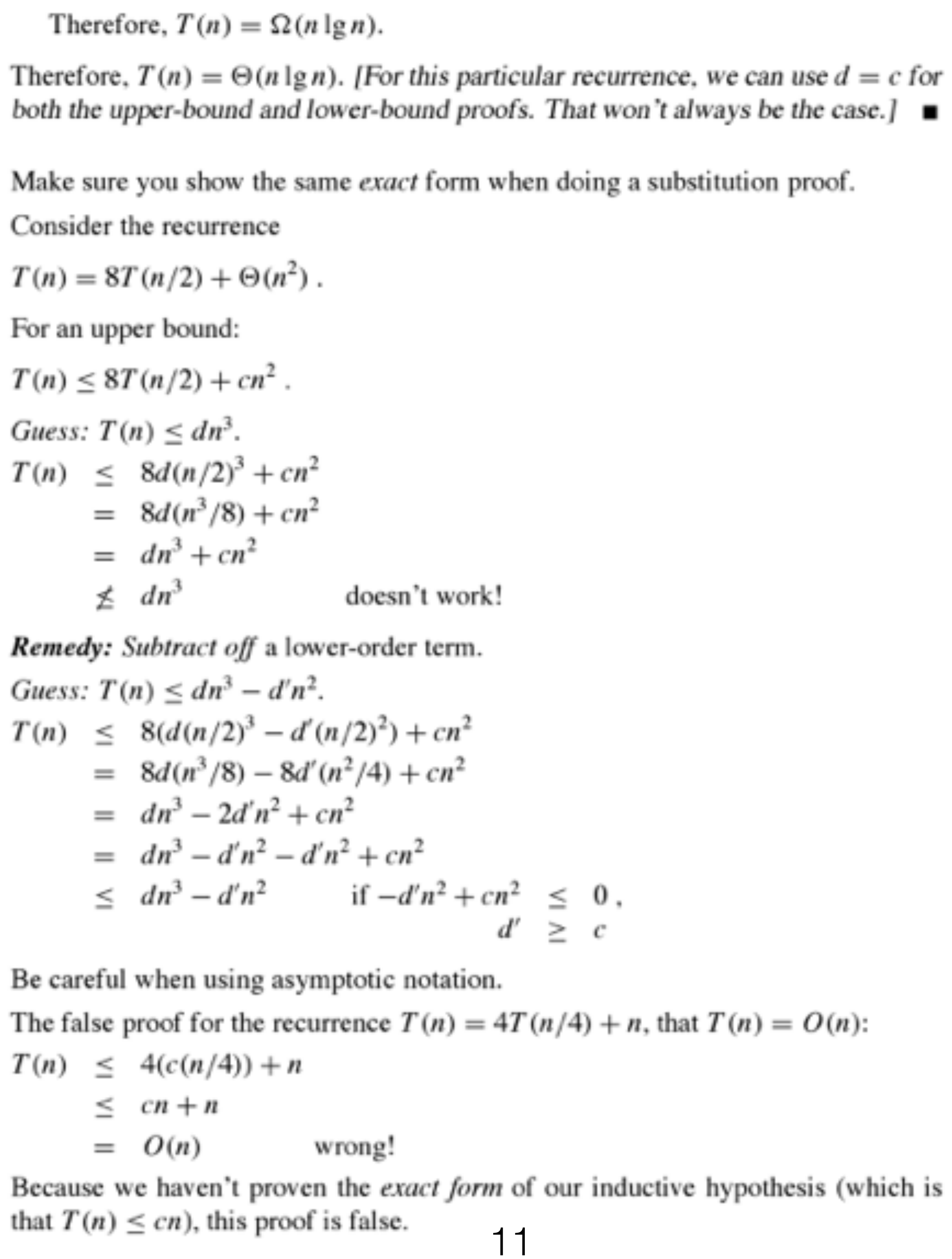
2. ***Lower bound:*** Write $T(n) \ge 2T(n/2) + cn$ for some positive constant c.

   Guess: $T(n) \ge dn \lg n$ for some positive constant d.

   Substitution:

$$\begin{aligned} T(n) &\ge 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\ge dn\lg n \qquad \text{if } -dn + cn \ge 0\,, \\ &\qquad\qquad\qquad\quad d \le c \end{aligned}$$

Substitution Method



Therefore, $T(n) = \Omega(n \lg n)$.

Therefore, $T(n) = \Theta(n \lg n)$. *[For this particular recurrence, we can use $d = c$ for both the upper-bound and lower-bound proofs. That won't always be the case.]* ■

Make sure you show the same *exact* form when doing a substitution proof.

Consider the recurrence

$T(n) = 8T(n/2) + \Theta(n^2)$.

For an upper bound:

$T(n) \leq 8T(n/2) + cn^2$.

Guess: $T(n) \leq dn^3$.

$$\begin{aligned} T(n) &\leq 8d(n/2)^3 + cn^2 \\ &= 8d(n^3/8) + cn^2 \\ &= dn^3 + cn^2 \\ &\not\leq dn^3 \qquad \text{doesn't work!} \end{aligned}$$

Remedy: *Subtract off* a lower-order term.

Guess: $T(n) \leq dn^3 - d'n^2$.

$$\begin{aligned} T(n) &\leq 8(d(n/2)^3 - d'(n/2)^2) + cn^2 \\ &= 8d(n^3/8) - 8d'(n^2/4) + cn^2 \\ &= dn^3 - 2d'n^2 + cn^2 \\ &= dn^3 - d'n^2 - d'n^2 + cn^2 \\ &\leq dn^3 - d'n^2 \qquad \text{if } -d'n^2 + cn^2 \leq 0 \,, \\ &\qquad\qquad\qquad\qquad\quad d' \geq c \end{aligned}$$

Be careful when using asymptotic notation.

The false proof for the recurrence $T(n) = 4T(n/4) + n$, that $T(n) = O(n)$:

$$\begin{aligned} T(n) &\leq 4(c(n/4)) + n \\ &\leq cn + n \\ &= O(n) \qquad \text{wrong!} \end{aligned}$$

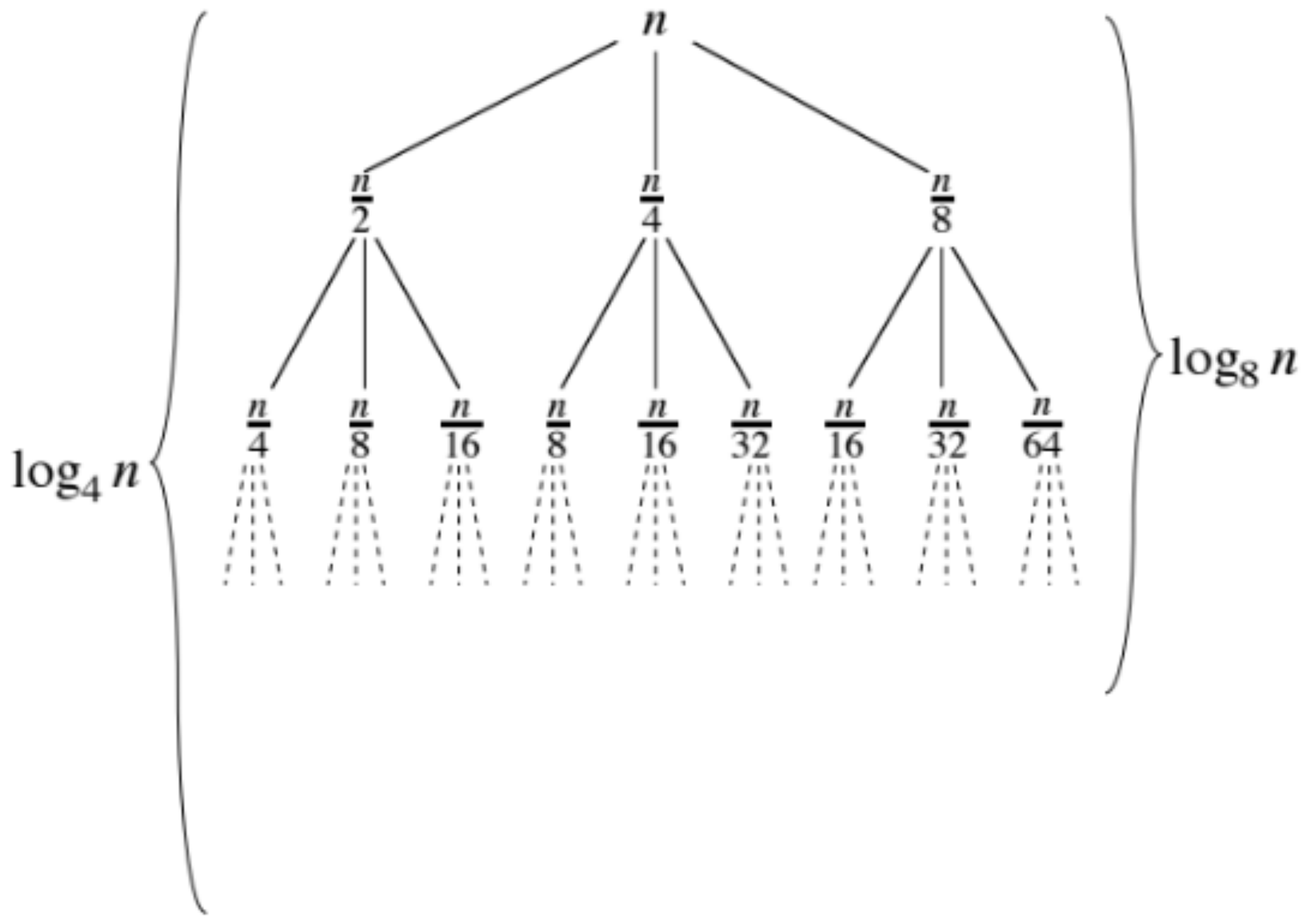
Because we haven't proven the *exact form* of our inductive hypothesis (which is that $T(n) \leq cn$), this proof is false.

Recursion Trees

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$

Using the recursion tree shown below, we get a guess of $T(n) = \Theta(n)$.

Recursion Trees



n
$\frac{n}{2}$ $\frac{n}{4}$ $\frac{n}{8}$
$\frac{n}{4}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{64}$
$\log_4 n$
$\log_8 n$

n

$$n\left(\frac{4+2+1}{8}\right) = \frac{7}{8}n$$

$$n\left(\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \frac{2}{32} + \frac{1}{64}\right)$$
$$= n\frac{16+16+12+4+1}{64}$$
$$= n\frac{49}{64} = \frac{7}{8}^2 n$$

$\vdots$

$$\sum_{i=1}^{\log n} \left(\frac{7}{8}\right)^i n = \Theta(n)$$

Recursion Trees

We use the substitution method to prove that $T(n) = O(n)$. Our inductive hypothesis is that $T(n) \le cn$ for some constant $c > 0$. We have

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + T(n/8) + n \\ &\le cn/2 + cn/4 + cn/8 + n \\ &= 7cn/8 + n \\ &= (1 + 7c/8)n \\ &\le cn \qquad \text{if } c \ge 8 . \end{aligned}$$

Therefore, $T(n) = O(n)$.

Showing that $T(n) = \Omega(n)$ is easy:

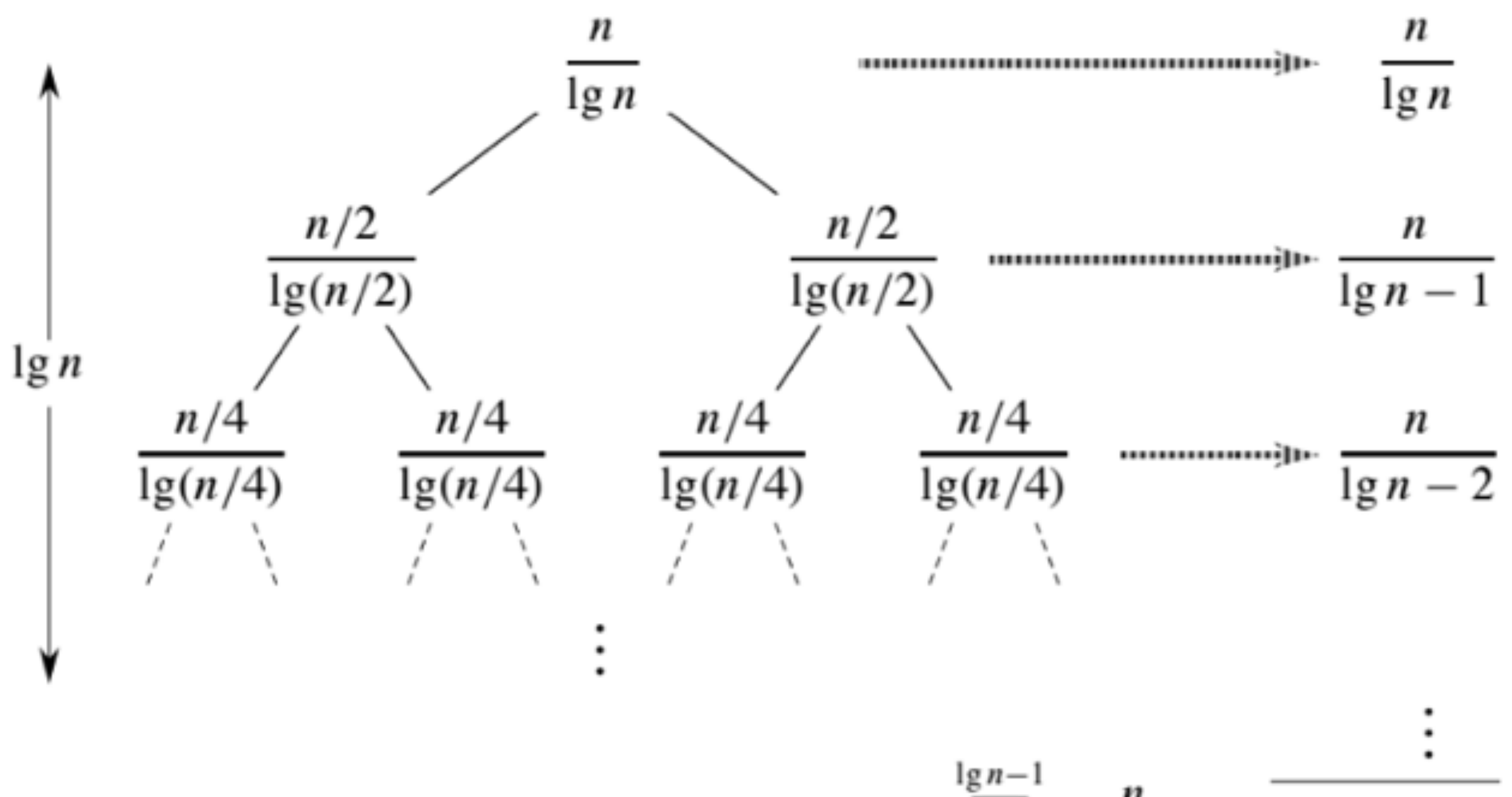
$$T(n) = T(n/2) + T(n/4) + T(n/8) + n \ge n .$$

Since $T(n) = O(n)$ and $T(n) = \Omega(n)$, we have that $T(n) = \Theta(n)$.

Recursion Trees

$T(n) = 2T(n/2) + n/\lg n$

We can get a guess by means of a recursion tree:



$$\sum_{i=0}^{\lg n-1} \frac{n}{\lg n - i} = \Theta(n \lg\lg n)$$

We get the sum on each level by observing that at depth i, we have 2^i nodes, each with a numerator of $n/2^i$ and a denominator of $\lg(n/2^i) = \lg n - i$, so that the cost at depth i is

Recursion Trees

$$2^i \cdot \frac{n/2^i}{\lg n - i} = \frac{n}{\lg n - i} .$$

The sum for all levels is

$$\begin{aligned}
\sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} &= n \sum_{i=1}^{\lg n} \frac{n}{i} \\
&= n \sum_{i=1}^{\lg n} 1/i \\
&= n \cdot \Theta(\lg \lg n) \quad \text{(by equation (A.7), the harmonic series)} \\
&= \Theta(n \lg \lg n) .
\end{aligned}$$

Recursion Trees

We can use this analysis as a guess that $T(n) = \Theta(n \lg\lg n)$. If we were to do a straight substitution proof, it would be rather involved. Instead, we will show by substitution that $T(n) \le n(1 + H_{\lfloor \lg n \rfloor})$ and $T(n) \ge n \cdot H_{\lceil \lg n \rceil}$, where H_k is the kth harmonic number: $H_k = 1/1 + 1/2 + 1/3 + \cdots + 1/k$. We also define $H_0 = 0$. Since $H_k = \Theta(\lg k)$, we have that $H_{\lfloor \lg n \rfloor} = \Theta(\lg \lfloor \lg n \rfloor) = \Theta(\lg\lg n)$ and $H_{\lceil \lg n \rceil} = \Theta(\lg \lceil \lg n \rceil) = \Theta(\lg\lg n)$. Thus, we will have that $T(n) = \Theta(n \lg\lg n)$.

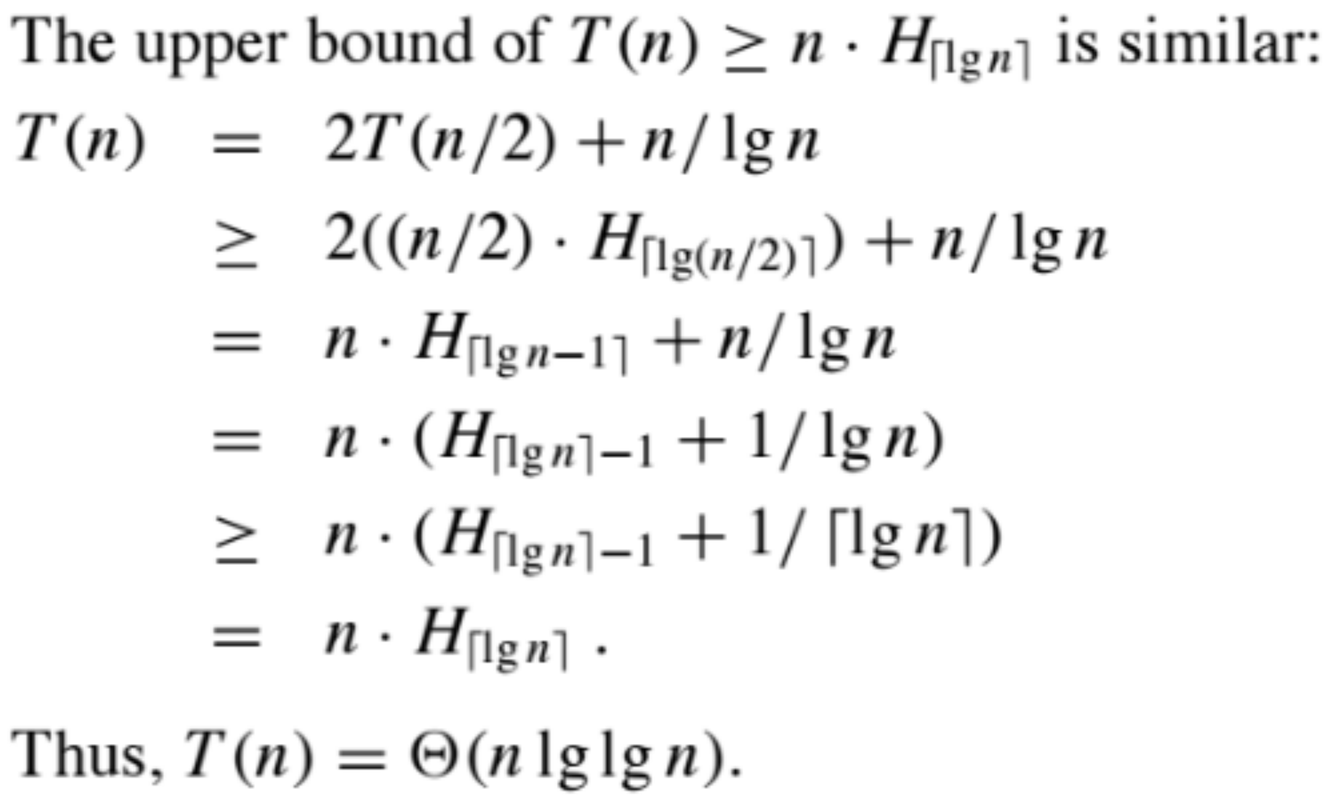
The base case for the proof is for $n = 1$, and we use $T(1) = 1$. Here, $\lg n = 0$, so that $\lg n = \lfloor \lg n \rfloor = \lceil \lg n \rceil$. Since $H_0 = 0$, we have $T(1) = 1 \le 1(1 + H_0)$ and $T(1) = 1 \ge 0 = 1 \cdot H_0$.

For the upper bound of $T(n) \le n(1 + H_{\lfloor \lg n \rfloor})$, we have

$$\begin{aligned}
T(n) &= 2T(n/2) + n/\lg n \\
&\le 2((n/2)(1 + H_{\lfloor \lg(n/2) \rfloor})) + n/\lg n \\
&= n(1 + H_{\lfloor \lg n - 1 \rfloor}) + n/\lg n \\
&= n(1 + H_{\lfloor \lg n \rfloor - 1} + 1/\lg n) \\
&\le n(1 + H_{\lfloor \lg n \rfloor - 1} + 1/\lfloor \lg n \rfloor) \\
&= n(1 + H_{\lfloor \lg n \rfloor}) ,
\end{aligned}$$

where the last line follows from the identity $H_k = H_{k-1} + 1/k$.

Recursion Trees

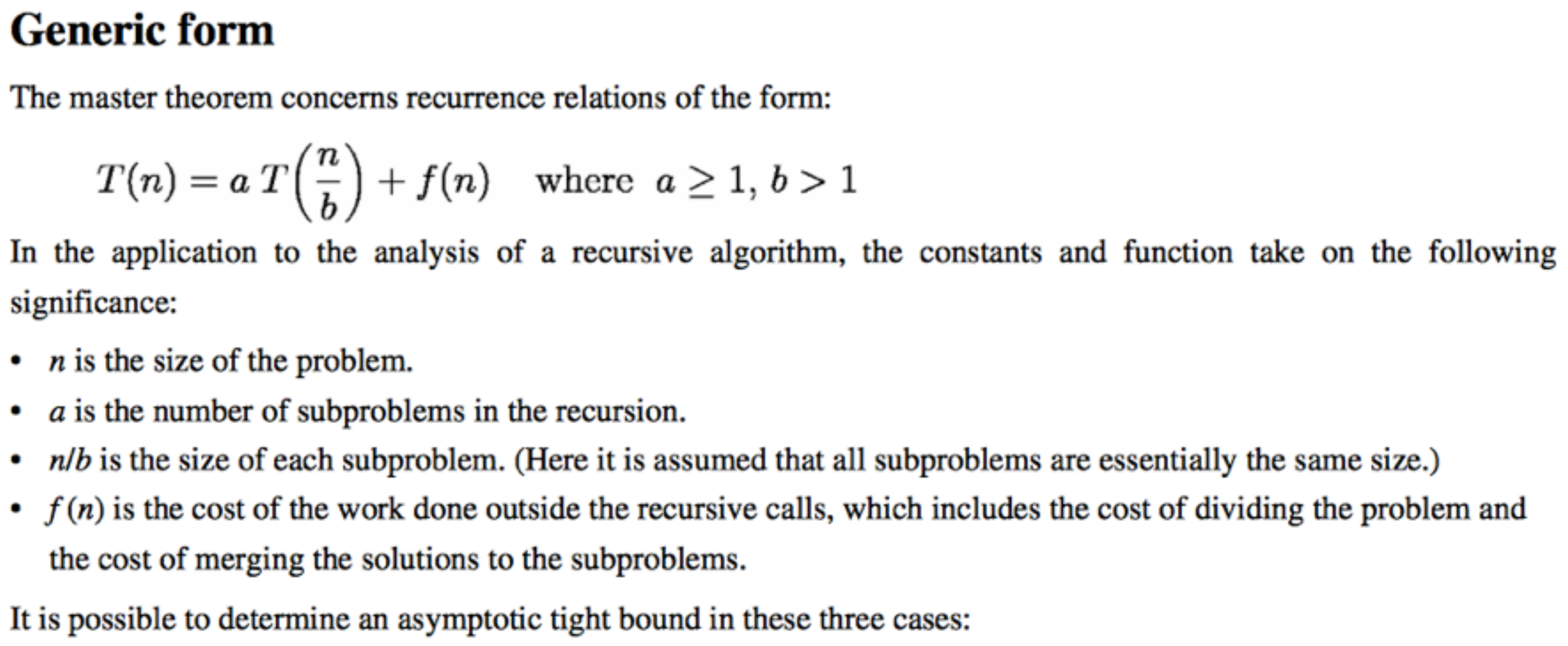


The upper bound of $T(n) \geq n \cdot H_{\lceil \lg n \rceil}$ is similar:

$$\begin{aligned}
T(n) &= 2T(n/2) + n/\lg n \\
&\geq 2((n/2) \cdot H_{\lceil \lg(n/2) \rceil}) + n/\lg n \\
&= n \cdot H_{\lceil \lg n - 1 \rceil} + n/\lg n \\
&= n \cdot (H_{\lceil \lg n \rceil - 1} + 1/\lg n) \\
&\geq n \cdot (H_{\lceil \lg n \rceil - 1} + 1/\lceil \lg n \rceil) \\
&= n \cdot H_{\lceil \lg n \rceil} \; .
\end{aligned}$$

Thus, $T(n) = \Theta(n \lg \lg n)$.

Master Teorem



Generic form

The master theorem concerns recurrence relations of the form:

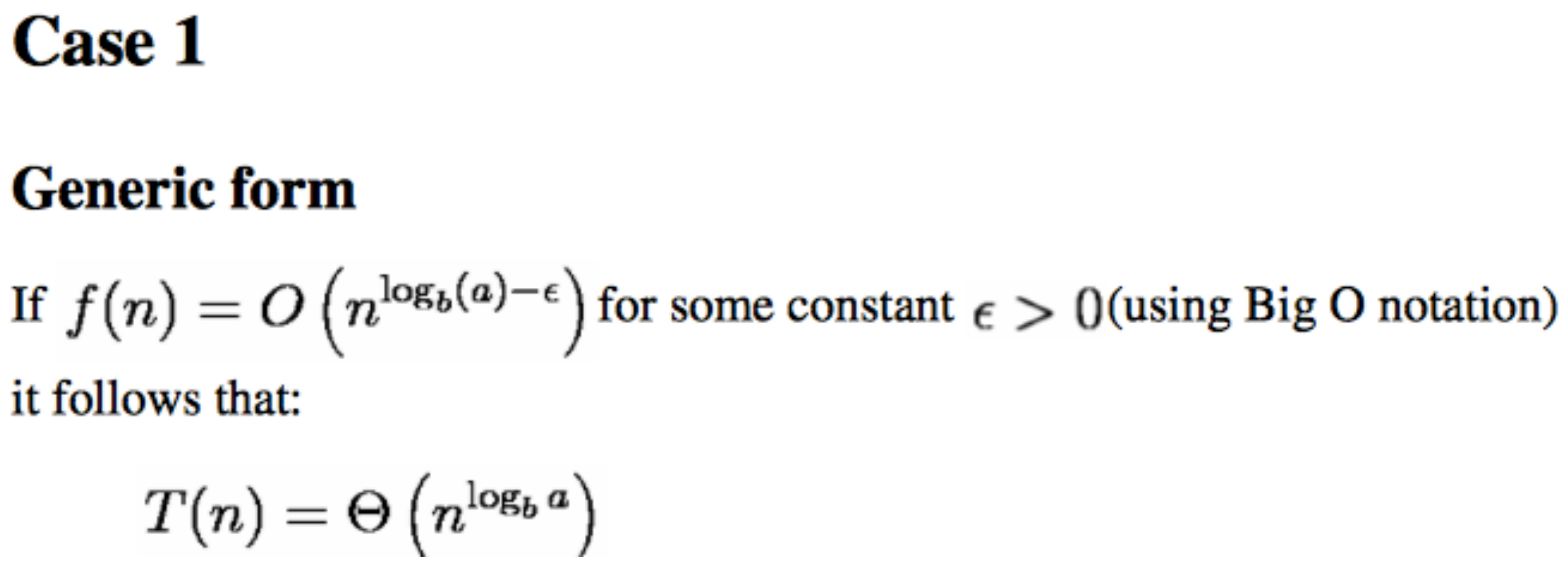
$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1,\, b > 1$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- *n* is the size of the problem.
- *a* is the number of subproblems in the recursion.
- *n*/*b* is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- *f* (*n*) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

It is possible to determine an asymptotic tight bound in these three cases:

Master Teorem



Case 1

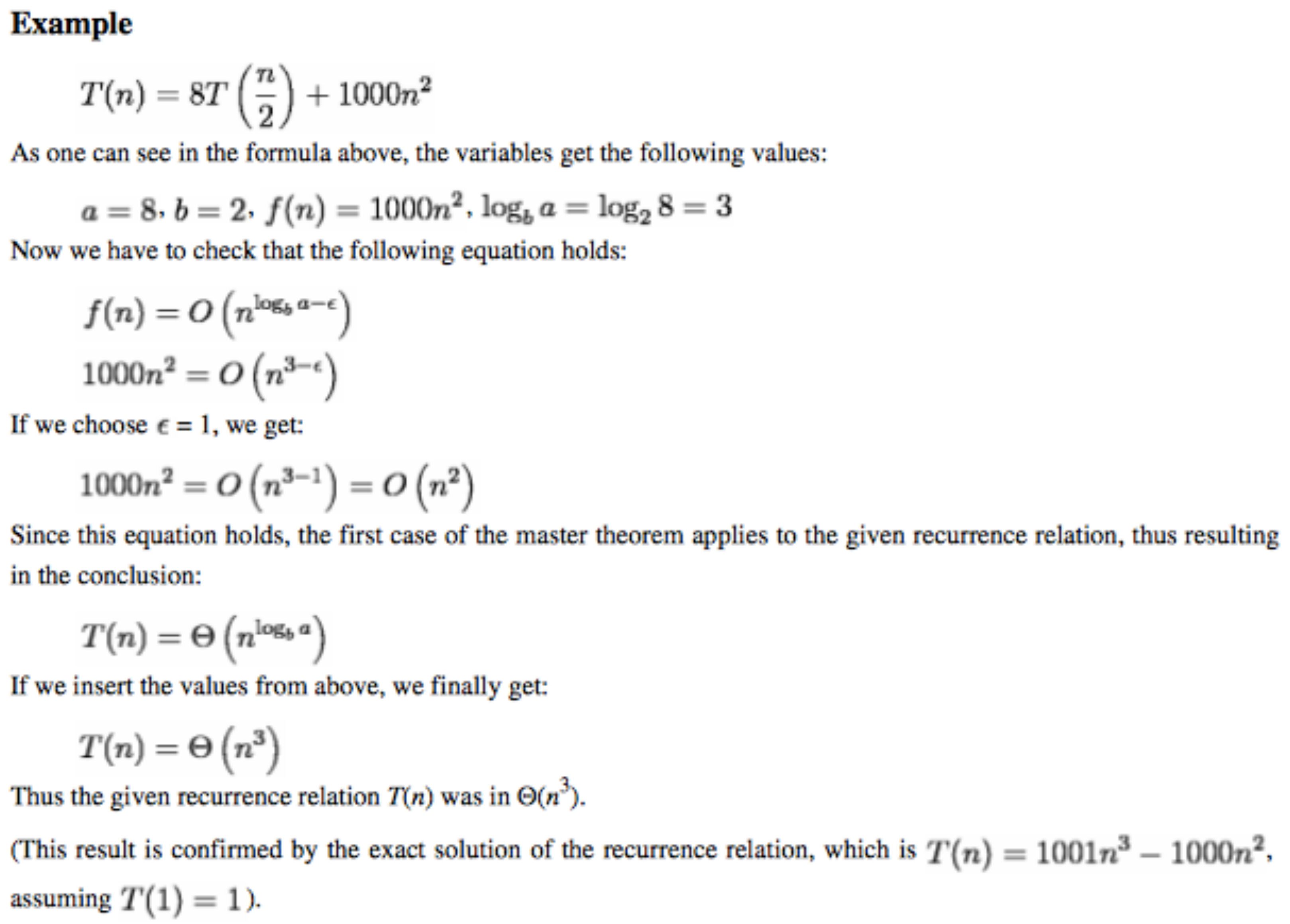
Generic form

If $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for some constant $\epsilon > 0$(using Big O notation)

it follows that:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Master Teorem



Example

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

As one can see in the formula above, the variables get the following values:

$$a = 8,\ b = 2,\ f(n) = 1000n^2,\ \log_b a = \log_2 8 = 3$$

Now we have to check that the following equation holds:

$$f(n) = O\left(n^{\log_b a - \epsilon}\right)$$

$$1000n^2 = O\left(n^{3-\epsilon}\right)$$

If we choose $\epsilon = 1$, we get:

$$1000n^2 = O\left(n^{3-1}\right) = O\left(n^2\right)$$

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

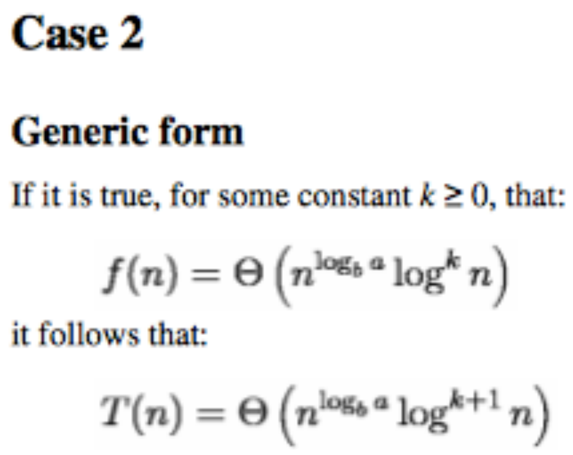
If we insert the values from above, we finally get:

$$T(n) = \Theta\left(n^3\right)$$

Thus the given recurrence relation $T(n)$ was in $\Theta(n^3)$.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 1001n^3 - 1000n^2$, assuming $T(1) = 1$).

Master Teorem



Case 2

Generic form

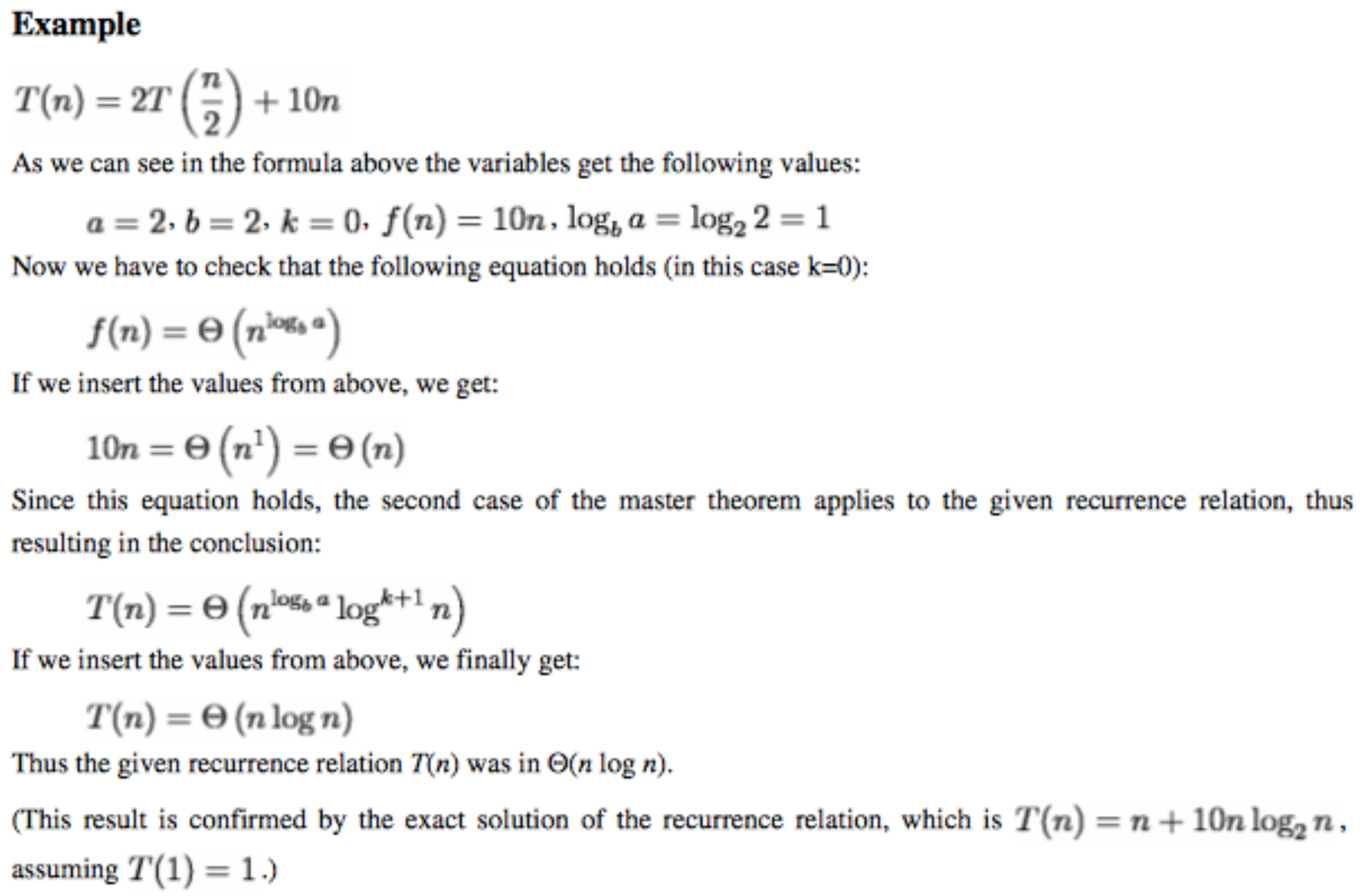
If it is true, for some constant $k \geq 0$, that:

$$f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$$

it follows that:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$

Master Teorem



Example

$T(n) = 2T\left(\frac{n}{2}\right) + 10n$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ k = 0,\ f(n) = 10n,\ \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds (in this case k=0):

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

If we insert the values from above, we get:

$$10n = \Theta\left(n^1\right) = \Theta(n)$$

Since this equation holds, the second case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$

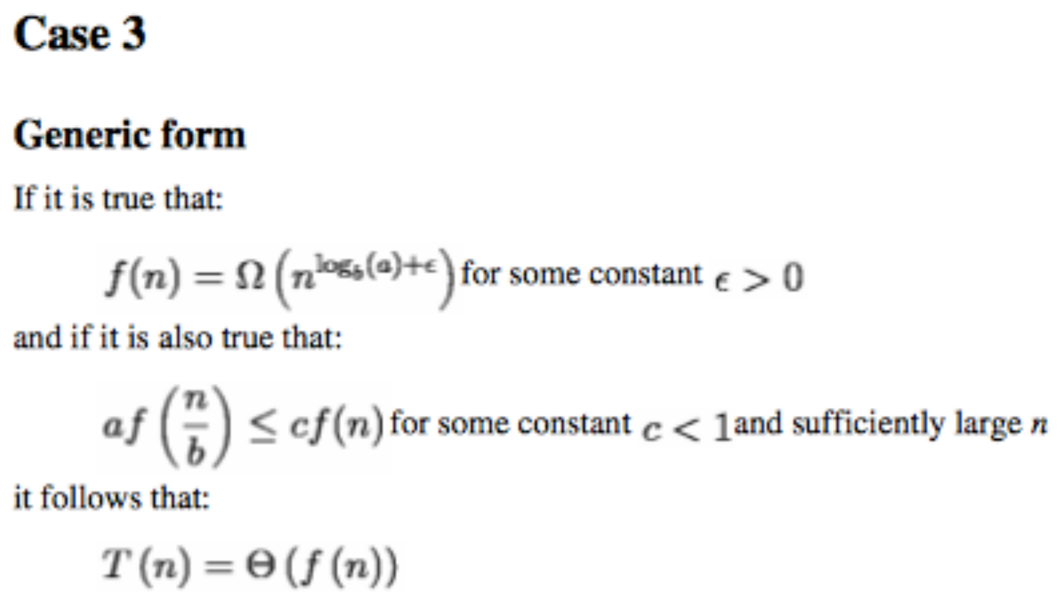
If we insert the values from above, we finally get:

$$T(n) = \Theta(n \log n)$$

Thus the given recurrence relation $T(n)$ was in $\Theta(n \log n)$.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = n + 10n \log_2 n$, assuming $T(1) = 1$.)

Master Teorem



Case 3

Generic form

If it is true that:

$$f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right) \text{ for some constant } \epsilon > 0$$

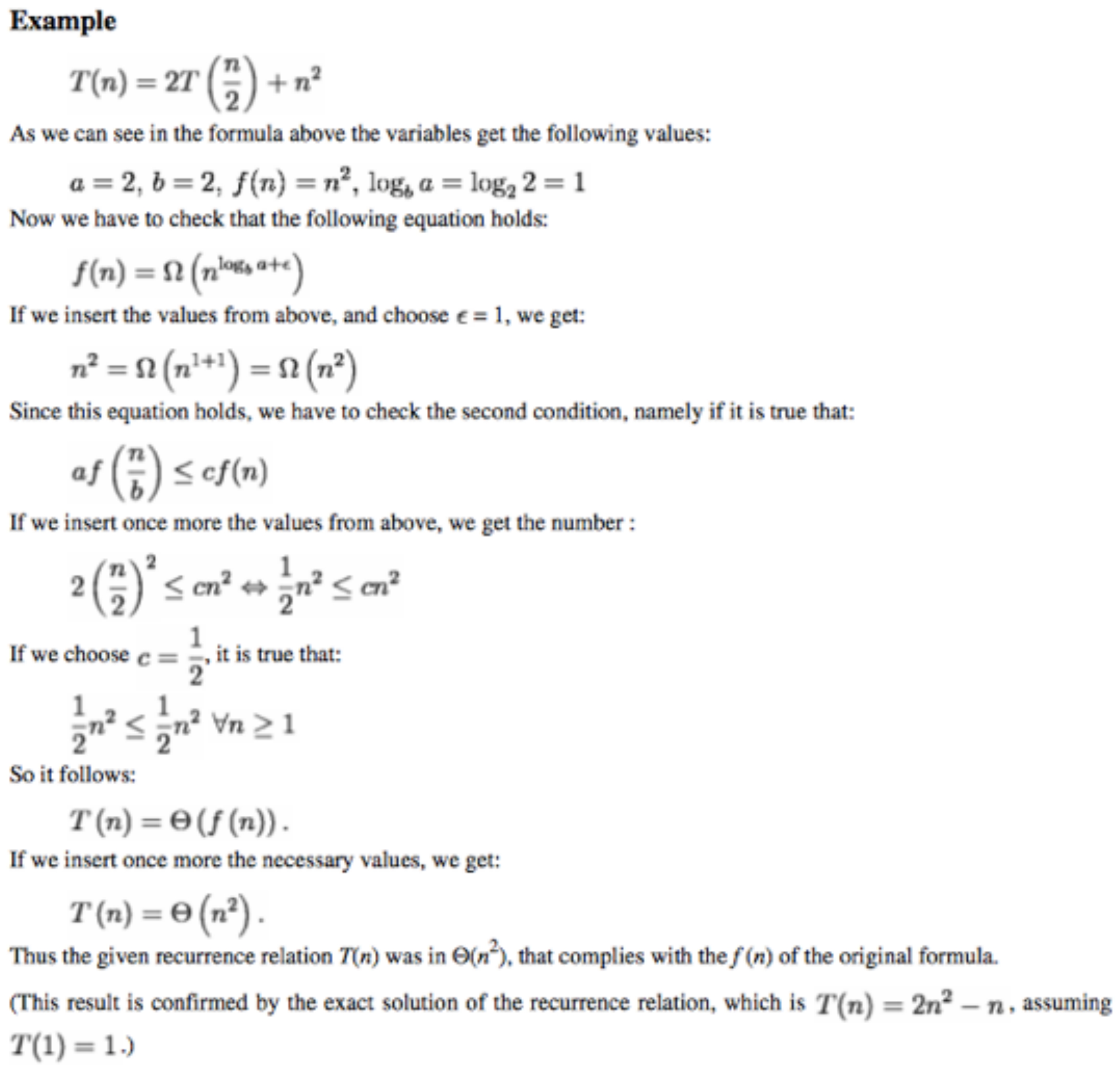
and if it is also true that:

$$af\left(\frac{n}{b}\right) \le cf(n) \text{ for some constant } c < 1 \text{ and sufficiently large } n$$

it follows that:

$$T(n) = \Theta(f(n))$$

Master Teorem



Example

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ f(n) = n^2,\ \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds:

$$f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$$

If we insert the values from above, and choose $\epsilon = 1$, we get:

$$n^2 = \Omega\left(n^{1+1}\right) = \Omega\left(n^2\right)$$

Since this equation holds, we have to check the second condition, namely if it is true that:

$$af\left(\frac{n}{b}\right) \le cf(n)$$

If we insert once more the values from above, we get the number :

$$2\left(\frac{n}{2}\right)^2 \le cn^2 \Leftrightarrow \frac{1}{2}n^2 \le cn^2$$

If we choose $c = \frac{1}{2}$, it is true that:

$$\frac{1}{2}n^2 \le \frac{1}{2}n^2\ \forall n \ge 1$$

So it follows:

$$T(n) = \Theta(f(n)).$$

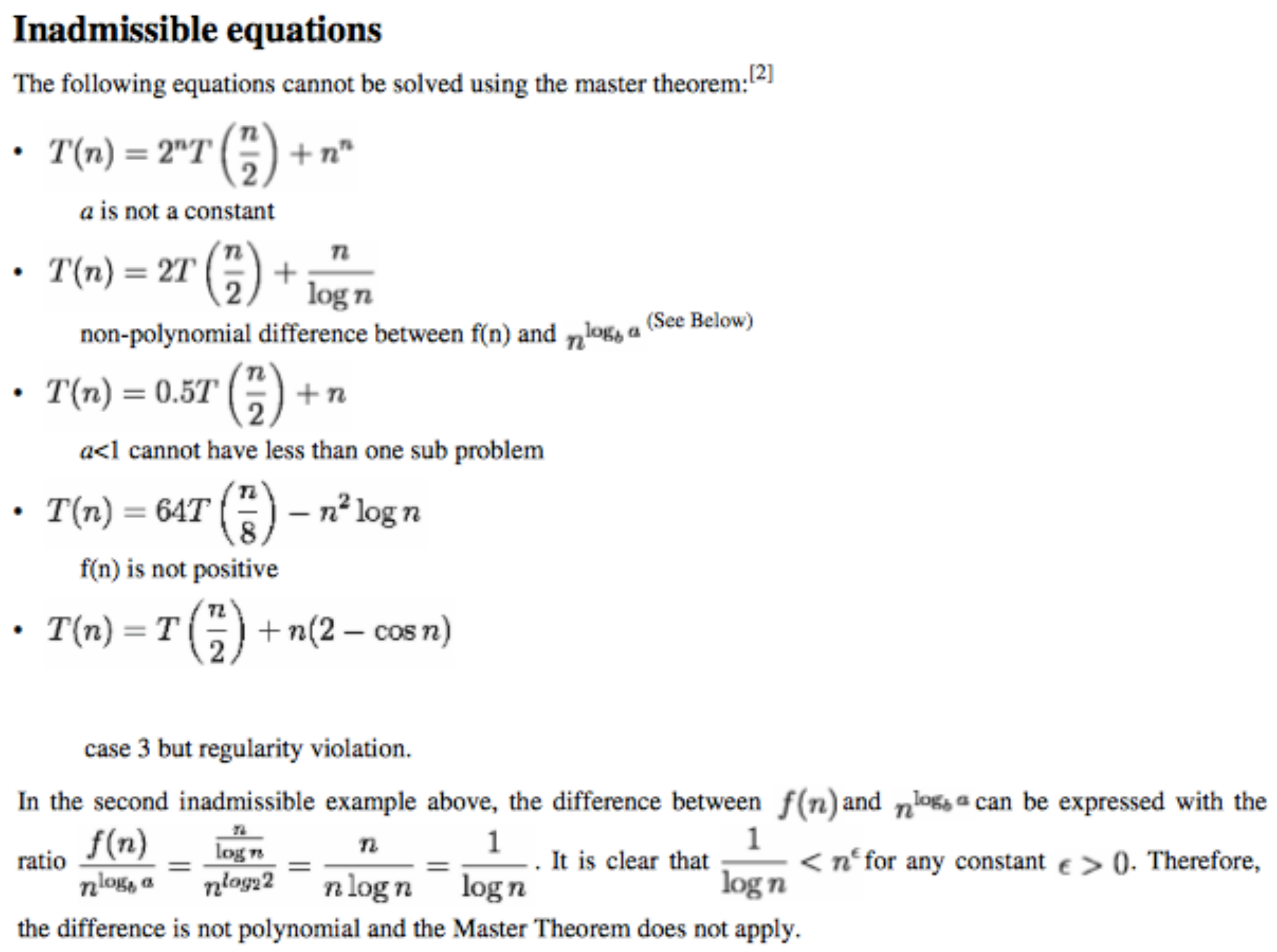
If we insert once more the necessary values, we get:

$$T(n) = \Theta\left(n^2\right).$$

Thus the given recurrence relation $T(n)$ was in $\Theta(n^2)$, that complies with the $f(n)$ of the original formula.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 2n^2 - n$, assuming $T(1) = 1$.)

Master Teorem



Inadmissible equations

The following equations cannot be solved using the master theorem:[2]

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$

  a is not a constant
- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

  non-polynomial difference between f(n) and $n^{\log_b a}$ (See Below)
- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

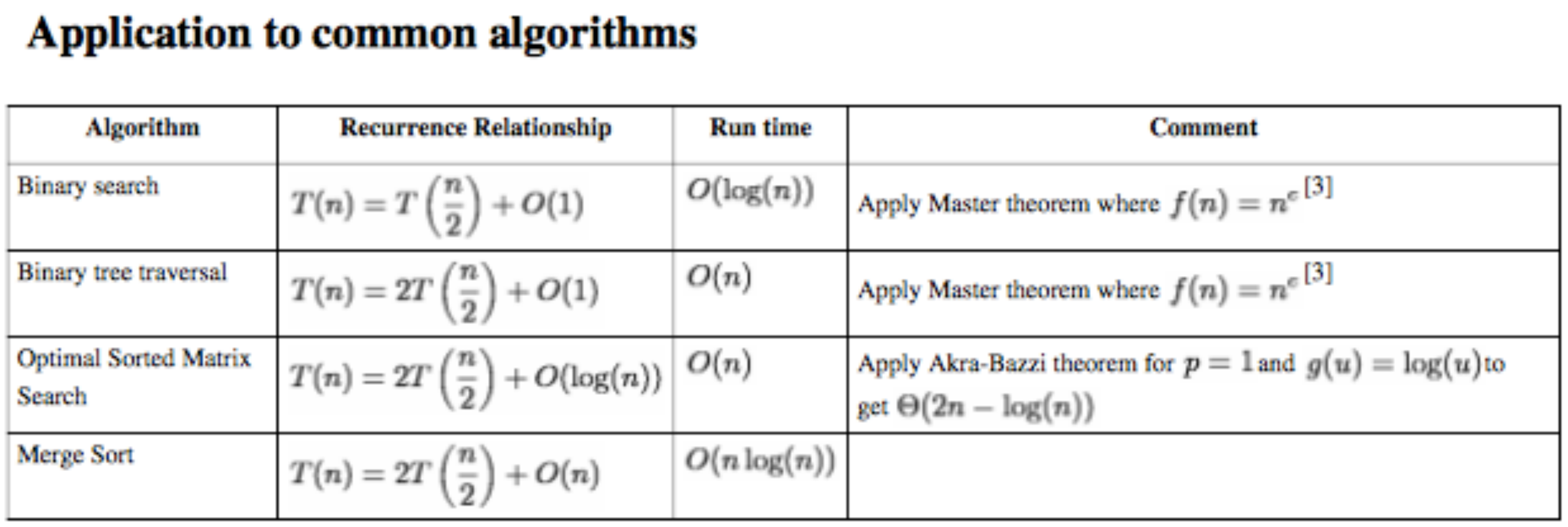
  a<1 cannot have less than one sub problem
- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

  f(n) is not positive
- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

  case 3 but regularity violation.

In the second inadmissible example above, the difference between $f(n)$ and $n^{\log_b a}$ can be expressed with the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{\frac{n}{\log n}}{n^{log_2 2}} = \frac{n}{n \log n} = \frac{1}{\log n}$. It is clear that $\frac{1}{\log n} < n^\epsilon$ for any constant $\epsilon > 0$. Therefore, the difference is not polynomial and the Master Theorem does not apply.

Master Teorem



Application to common algorithms

Algorithm	Recurrence Relationship	Run time	Comment
Binary search	$T(n) = T\left(\frac{n}{2}\right) + O(1)$	$O(\log(n))$	Apply Master theorem where $f(n) = n^c$ [3]
Binary tree traversal	$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$	$O(n)$	Apply Master theorem where $f(n) = n^c$ [3]
Optimal Sorted Matrix Search	$T(n) = 2T\left(\frac{n}{2}\right) + O(\log(n))$	$O(n)$	Apply Akra-Bazzi theorem for $p = 1$ and $g(u) = \log(u)$ to get $\Theta(2n - \log(n))$
Merge Sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n\log(n))$	

Master Teorem Slides Reference

http://www.saylor.org/site/wp-content/uploads/2011/06/Master-theorem.pdf

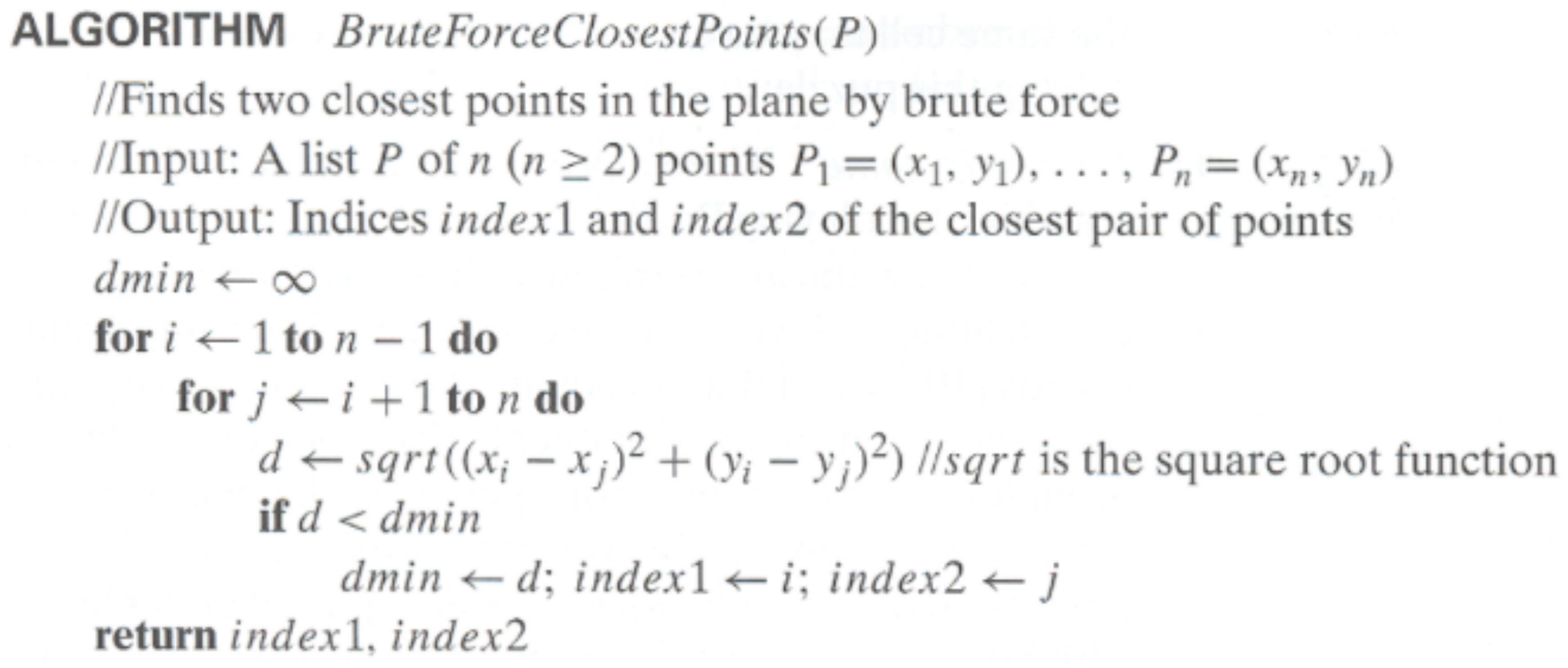
Brute Force Algorithms

- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved
- Examples:
  - Computing an (a > 0, n a nonnegative integer) by repeated multiplication
  - Computing n! by repeated multiplication
  - Multiplying two matrices following the definition
  - Searching for a key in a list sequentially

Brute Force

- Closest-Pair Problem
- Find the two closest points in a set of n points (in the two-dimensional Cartesian plane).
- Brute Force Algorithm
    - Compute the distance between every pair of distinct points and return the indices of the points for which the distance is the smallest.

Brute Force



```
ALGORITHM  BruteForceClosestPoints(P)
    //Finds two closest points in the plane by brute force
    //Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
    //Output: Indices index1 and index2 of the closest pair of points
    $dmin \leftarrow \infty$
    for $i \leftarrow 1$ to $n - 1$ do
        for $j \leftarrow i + 1$ to $n$ do
            $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //sqrt is the square root function
            if $d < dmin$
                $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
    return index1, index2
```

Exhaustive Search

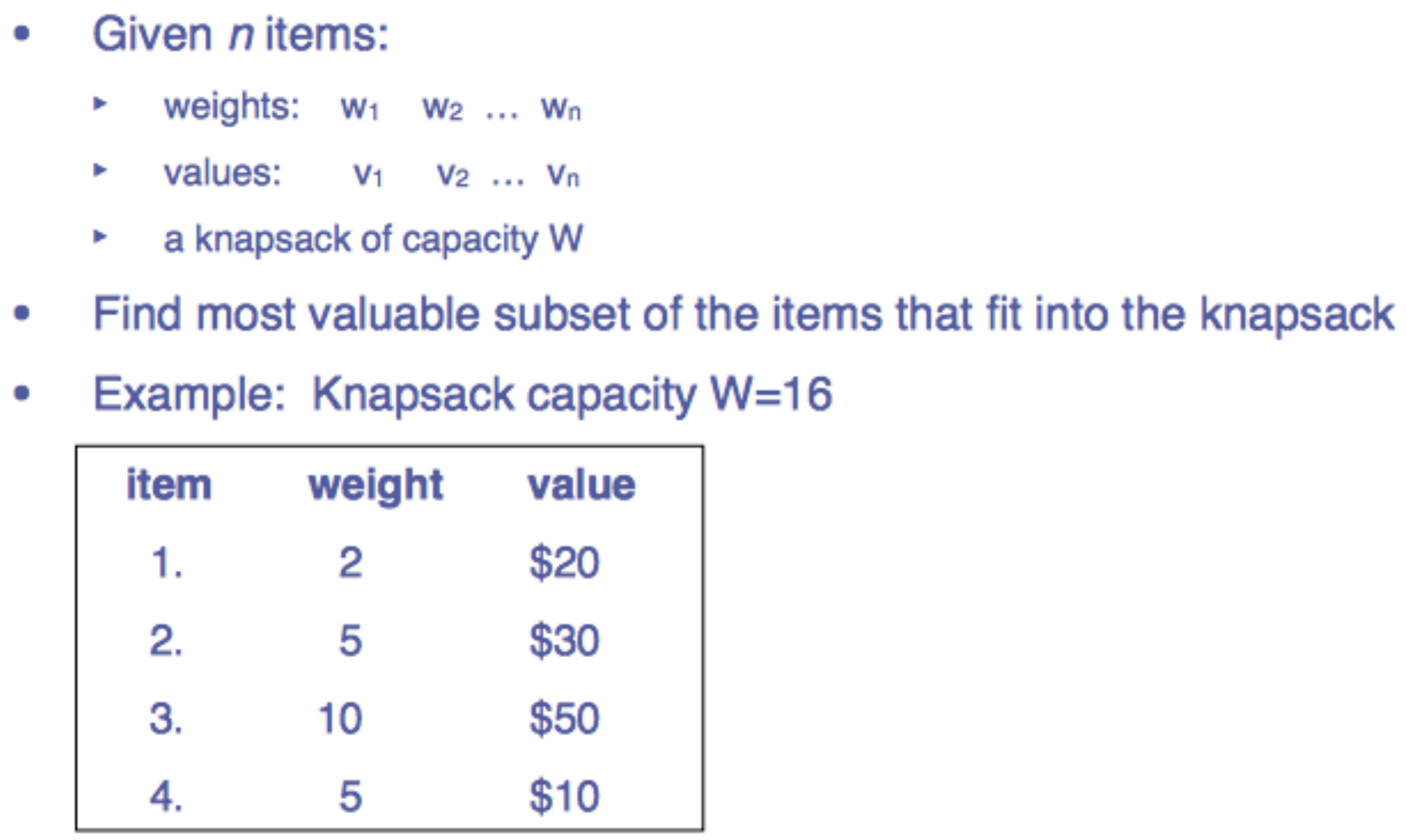
- A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.
- Method
  - Generate a list of all potential solutions to the problem in a systematic manner
  - Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimisation problem, keeping track of the best one found so far
  - when search ends, announce the solution(s) found

Exhaustive Search

- Traveling Salesman Problem
- Knapsack Problem
- The Assignment Problem

Exhaustive Search

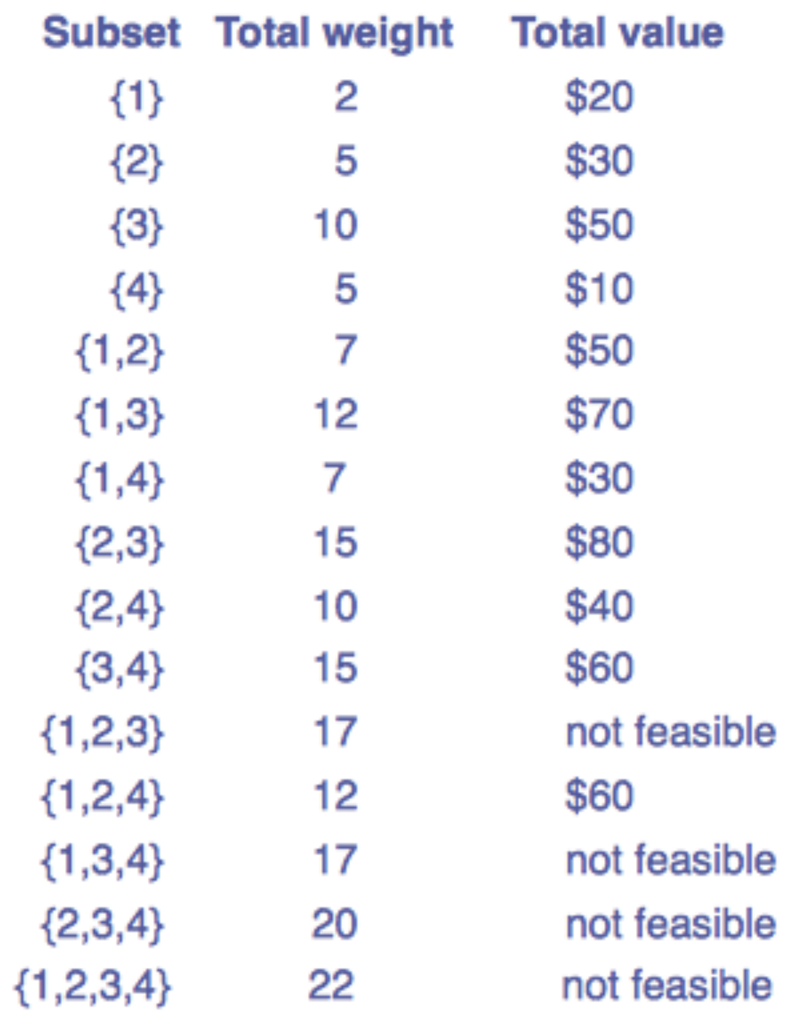
Knapsack Problem



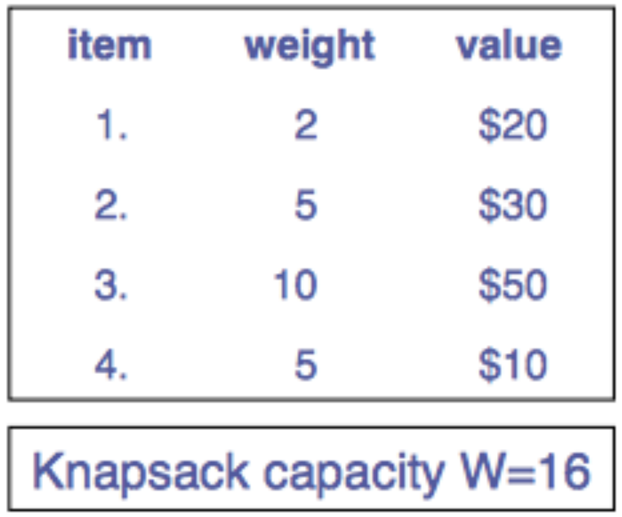
- Given *n* items:
  - weights: w_1 w_2 … w_n
  - values: v_1 v_2 … v_n
  - a knapsack of capacity W
- Find most valuable subset of the items that fit into the knapsack
- Example: Knapsack capacity W=16

item	weight	value
1.	2	$20
2.	5	$30
3.	10	$50
4.	5	$10

Exhaustive Search



Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible



item	weight	value
1.	2	$20
2.	5	$30
3.	10	$50
4.	5	$10

Knapsack capacity W=16

Brute Force - Exhaustive Search Slides Reference

http://web.cecs.pdx.edu/~black/cs350/Lectures/lec06-Exhaustive%20Search.pdf

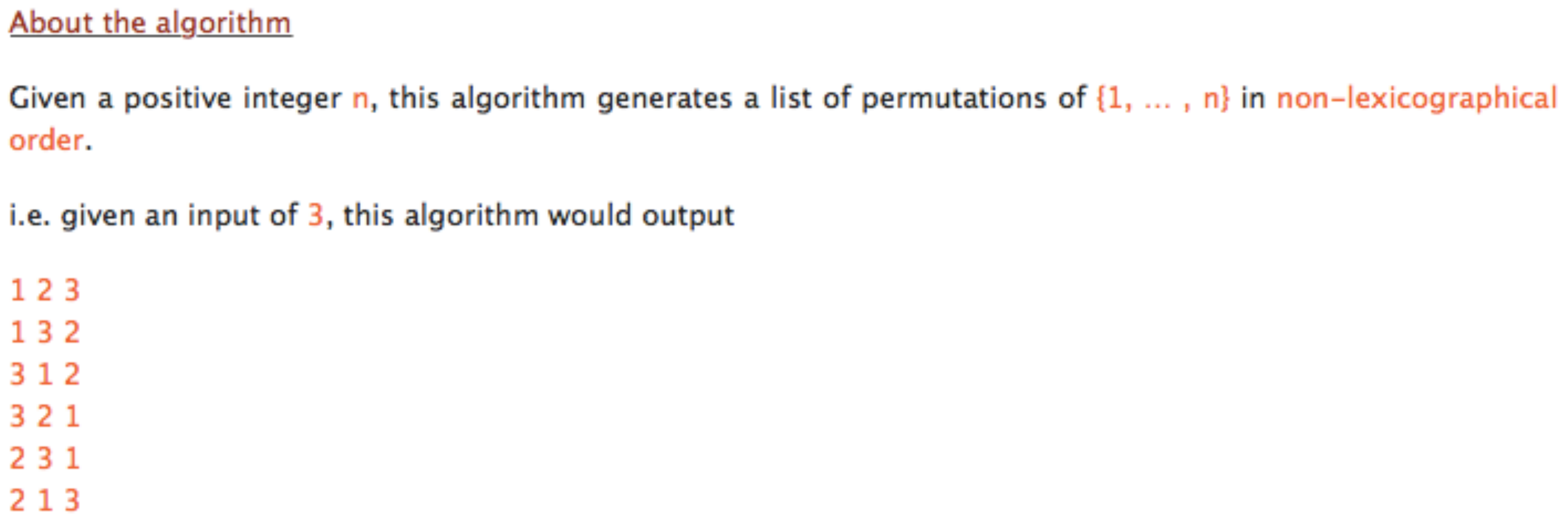
Decrease and Conquer Algorithms

- The Decrease-and-Conquer paradigm relies on a relation between an instance of the problem and a smaller instance of the same problem. It may happen that a given problem can be solved by decrease-by-constant as well as decrease-by-factor versions of the paradigm, for example computing an. While the algorithms in this group are usually described recursively, the implementations can be either recursive or iterative. The iterative implementations may require more coding effort, however they avoid the overload that accompanies recursion.

Decrease and Conquer Algorithms

- Steinhaus Johnson Trotter permutation algorithm

Decrease and Conquer Algorithms



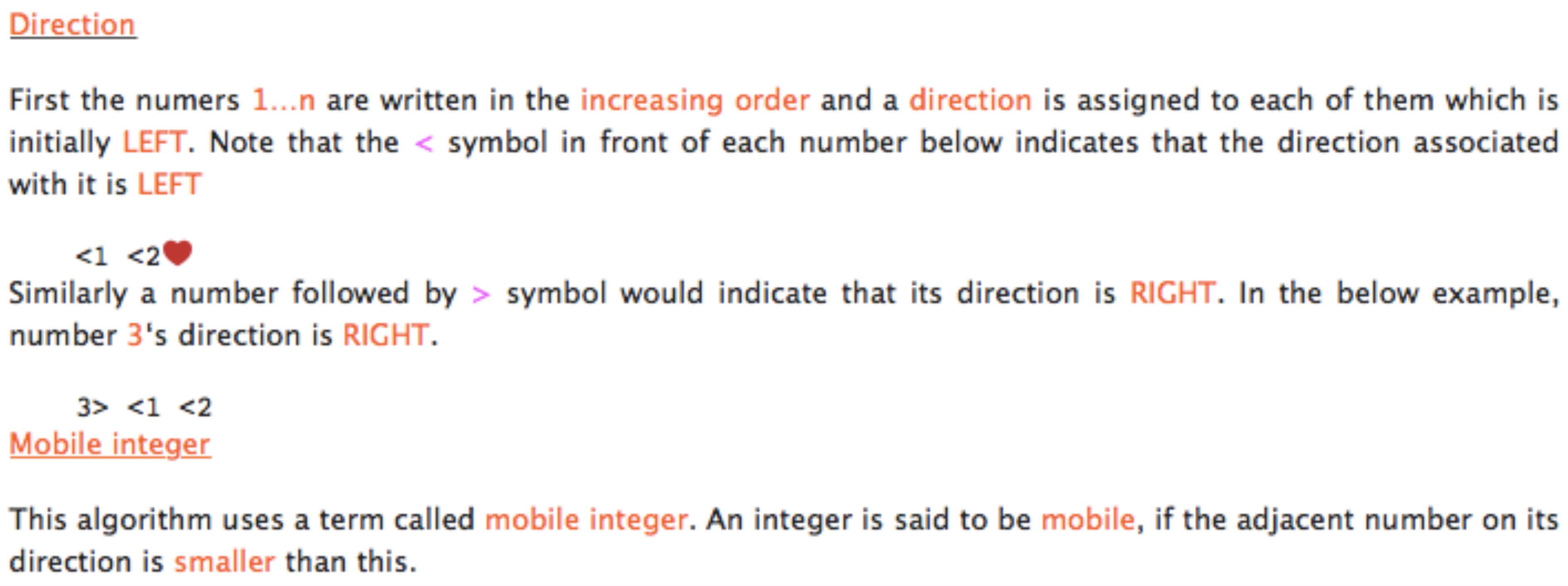
About the algorithm

Given a positive integer n, this algorithm generates a list of permutations of {1, … , n} in non-lexicographical order.

i.e. given an input of 3, this algorithm would output

```
1 2 3
1 3 2
3 1 2
3 2 1
2 3 1
2 1 3
```

Decrease and Conquer Algorithms



Direction

First the numers 1...n are written in the increasing order and a direction is assigned to each of them which is initially LEFT. Note that the < symbol in front of each number below indicates that the direction associated with it is LEFT

```
    <1 <2❤
```

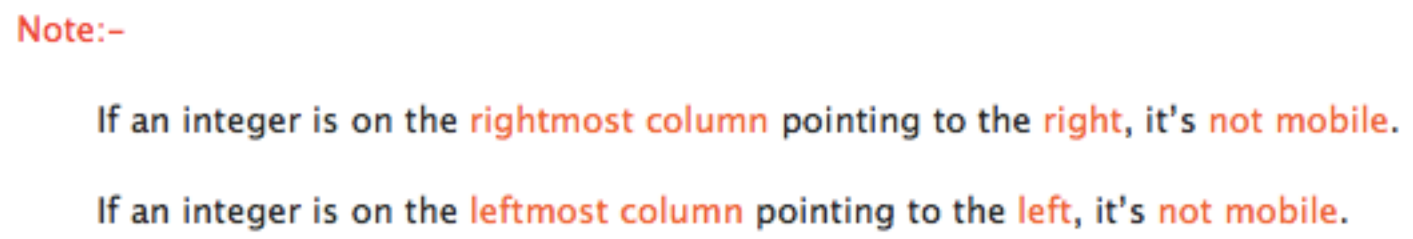
Similarly a number followed by > symbol would indicate that its direction is RIGHT. In the below example, number 3's direction is RIGHT.

```
    3> <1 <2
```

Mobile integer

This algorithm uses a term called mobile integer. An integer is said to be mobile, if the adjacent number on its direction is smaller than this.

Decrease and Conquer Algorithms

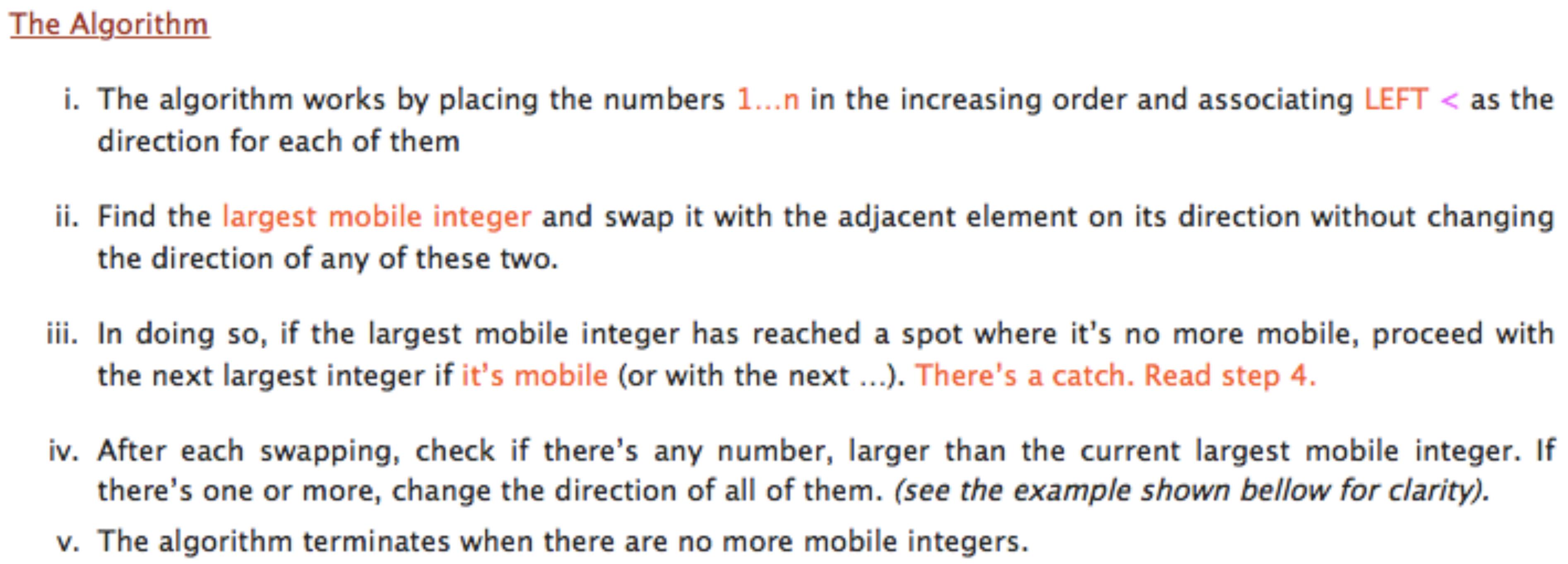


Note:-

If an integer is on the rightmost column pointing to the right, it's not mobile.

If an integer is on the leftmost column pointing to the left, it's not mobile.

Decrease and Conquer Algorithms



The Algorithm

i. The algorithm works by placing the numbers 1…n in the increasing order and associating LEFT < as the direction for each of them

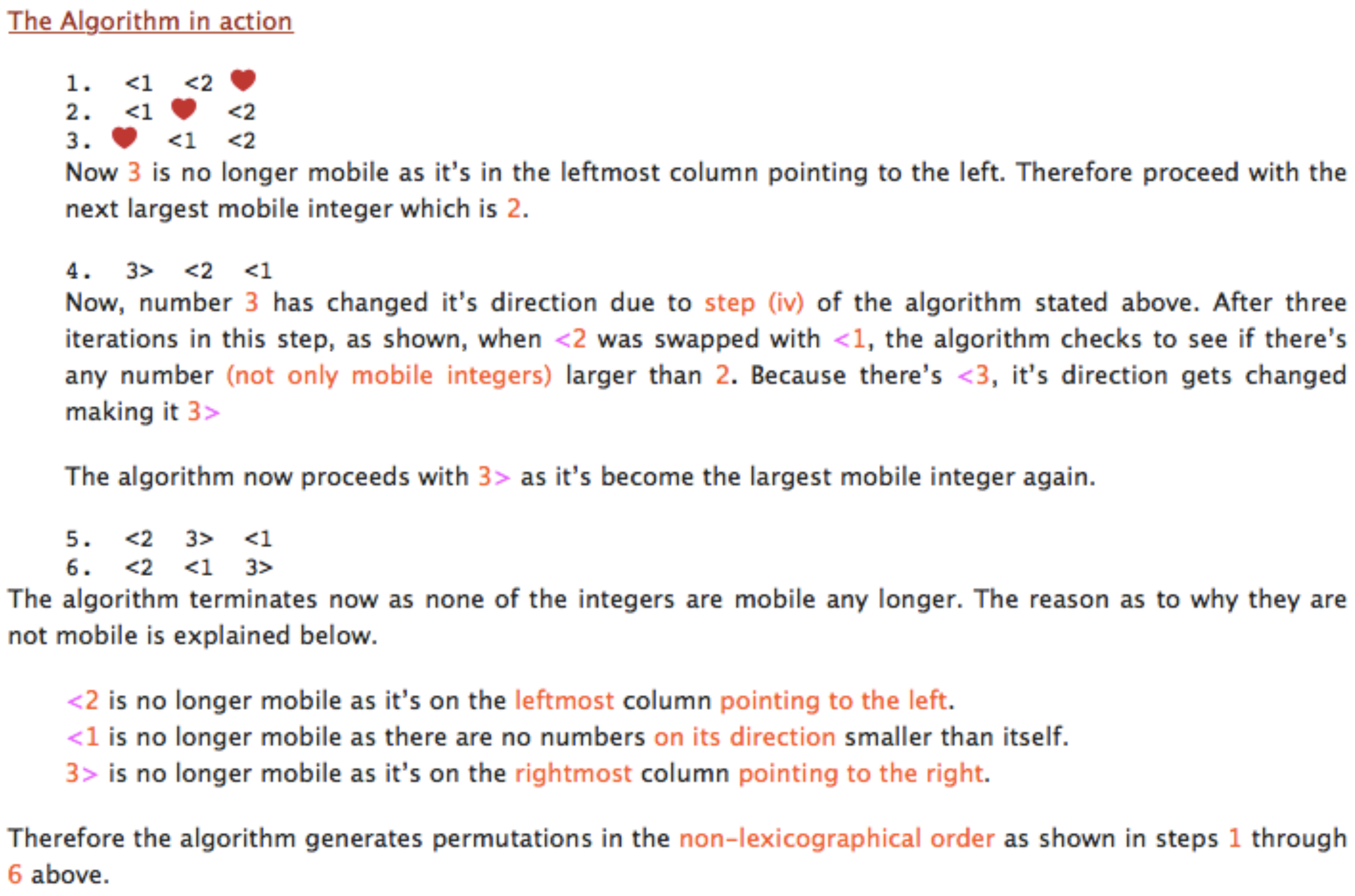
ii. Find the largest mobile integer and swap it with the adjacent element on its direction without changing the direction of any of these two.

iii. In doing so, if the largest mobile integer has reached a spot where it's no more mobile, proceed with the next largest integer if it's mobile (or with the next …). There's a catch. Read step 4.

iv. After each swapping, check if there's any number, larger than the current largest mobile integer. If there's one or more, change the direction of all of them. *(see the example shown bellow for clarity).*

v. The algorithm terminates when there are no more mobile integers.

Decrease and Conquer Algorithms



The Algorithm in action

```
1.  <1  <2  ❤
2.  <1  ❤  <2
3.  ❤  <1  <2
```

Now 3 is no longer mobile as it's in the leftmost column pointing to the left. Therefore proceed with the next largest mobile integer which is 2.

```
4.  3>  <2  <1
```

Now, number 3 has changed it's direction due to step (iv) of the algorithm stated above. After three iterations in this step, as shown, when <2 was swapped with <1, the algorithm checks to see if there's any number (not only mobile integers) larger than 2. Because there's <3, it's direction gets changed making it 3>

The algorithm now proceeds with 3> as it's become the largest mobile integer again.

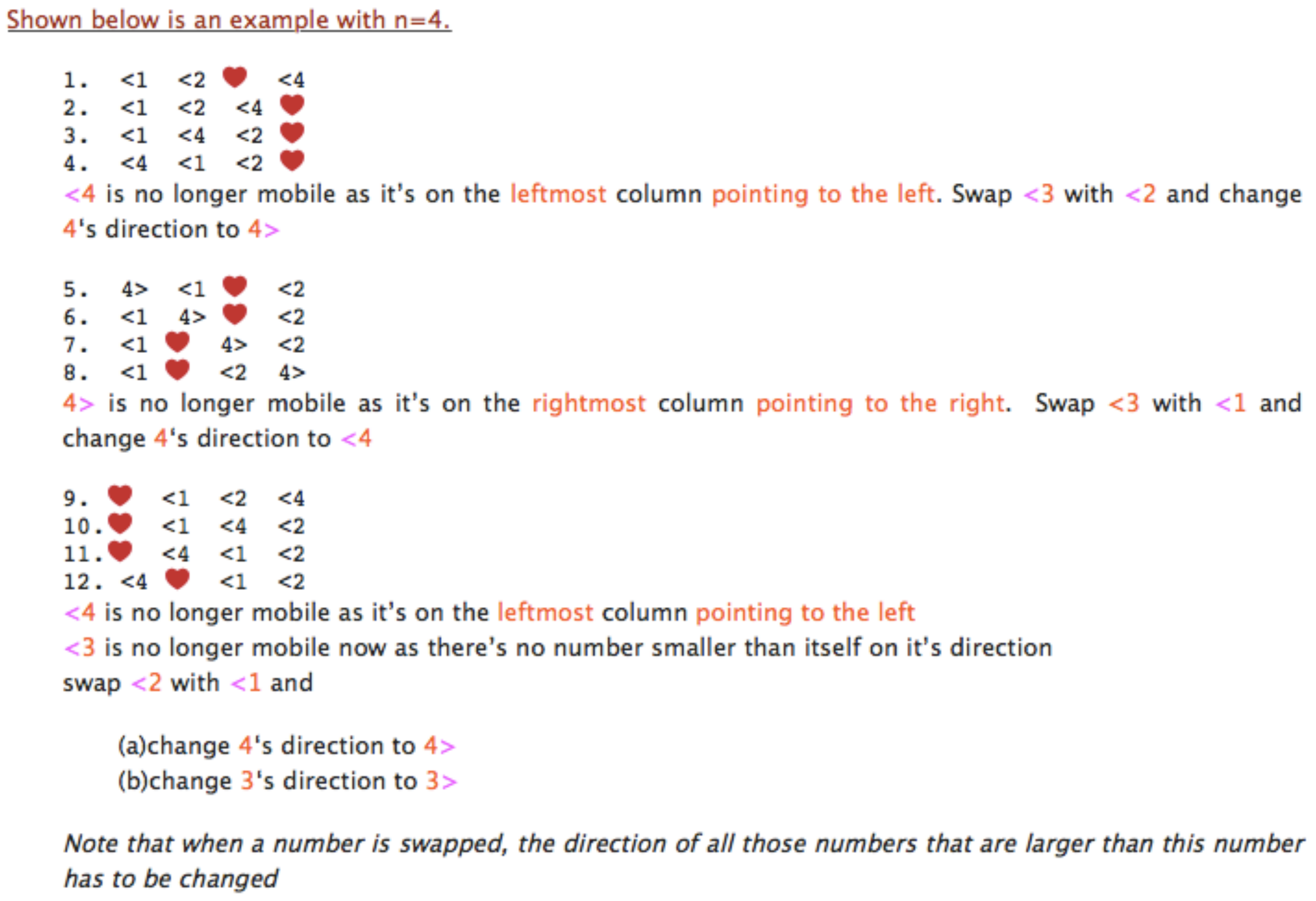
```
5.  <2  3>  <1
6.  <2  <1  3>
```

The algorithm terminates now as none of the integers are mobile any longer. The reason as to why they are not mobile is explained below.

<2 is no longer mobile as it's on the leftmost column pointing to the left.
<1 is no longer mobile as there are no numbers on its direction smaller than itself.
3> is no longer mobile as it's on the rightmost column pointing to the right.

Therefore the algorithm generates permutations in the non-lexicographical order as shown in steps 1 through 6 above.

Decrease and Conquer Algorithms



Shown below is an example with n=4.

```
1.   <1   <2   ❤   <4
2.   <1   <2   <4   ❤
3.   <1   <4   <2   ❤
4.   <4   <1   <2   ❤
```

<4 is no longer mobile as it's on the leftmost column pointing to the left. Swap <3 with <2 and change 4's direction to 4>

```
5.   4>   <1   ❤   <2
6.   <1   4>   ❤   <2
7.   <1   ❤   4>   <2
8.   <1   ❤   <2   4>
```

4> is no longer mobile as it's on the rightmost column pointing to the right. Swap <3 with <1 and change 4's direction to <4

```
9.   ❤   <1   <2   <4
10.  ❤   <1   <4   <2
11.  ❤   <4   <1   <2
12.  <4   ❤   <1   <2
```

<4 is no longer mobile as it's on the leftmost column pointing to the left

<3 is no longer mobile now as there's no number smaller than itself on it's direction

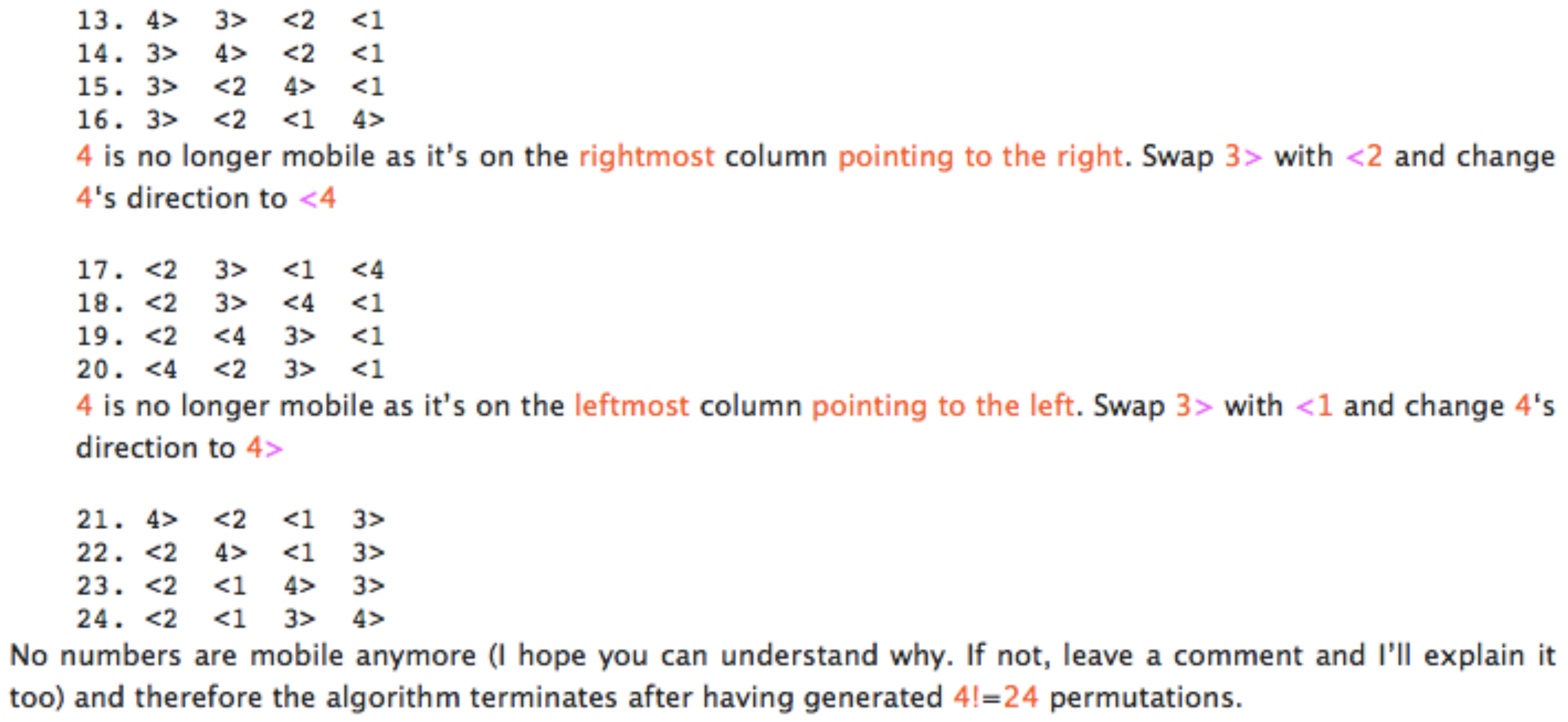
swap <2 with <1 and

(a)change 4's direction to 4>

(b)change 3's direction to 3>

Note that when a number is swapped, the direction of all those numbers that are larger than this number has to be changed

Decrease and Conquer Algorithms



```
13. 4>  3>  <2  <1
14. 3>  4>  <2  <1
15. 3>  <2  4>  <1
16. 3>  <2  <1  4>
```

4 is no longer mobile as it's on the rightmost column pointing to the right. Swap 3> with <2 and change 4's direction to <4

```
17. <2  3>  <1  <4
18. <2  3>  <4  <1
19. <2  <4  3>  <1
20. <4  <2  3>  <1
```

4 is no longer mobile as it's on the leftmost column pointing to the left. Swap 3> with <1 and change 4's direction to 4>

```
21. 4>  <2  <1  3>
22. <2  4>  <1  3>
23. <2  <1  4>  3>
24. <2  <1  3>  4>
```

No numbers are mobile anymore (I hope you can understand why. If not, leave a comment and I'll explain it too) and therefore the algorithm terminates after having generated 4!=24 permutations.

Decrease and Conquer Algorithms Slides Reference

http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Levitin/PDF/L07-PermutationsSubsets.pdf

https://tropenhitze.wordpress.com/2010/01/25/steinhaus-johnson-trotter-permutation-algorithm-explained-and-implemented-in-java/

Problems and Solutions

- Consider the problem of checking whether the sum of the elements in an array A[1..n] is even..
- a) What is the basic operation in this problem?
- b) Design a brute-force algorithm. Analyse the number of basic operations performed.
- c) Design a divide-and-conquer algorithm by dividing the problem into two almost equal size problems. Analyse the number of basic operations performed.
- d) Compare two algorithms in a and b, which one is more efficient?

Problems and Solutions

- a) Basic operation is addition
- b) Algorithm checkevensum (A[1..n])

```
//input A[1..n]
//output 1 if sum is even 0 otherwise

sum = A[1] % 2
for i=2 to n
  sum = sum XOR A[i]

if((sum % 2 ) == 0)
  return 0
else
  return 1

T(n) : Number of additions  = n-1
```

Problems and Solutions

- c) Algorithm checkevensum(A[l,r])

```
//input A[l..r]
//output 1 if sum is even 0 otherwise
if l>r
  return 1

m = (l + r ) / 2   // floor

a = checkevensum (A[l , m])
b = checkevensum (A[m+1, l])

return ((a+b) % 2 == 0)
```

- T(n) : number of additions
  $T(n) = 2\,T(n/2) + 1$

Problems and Soltions

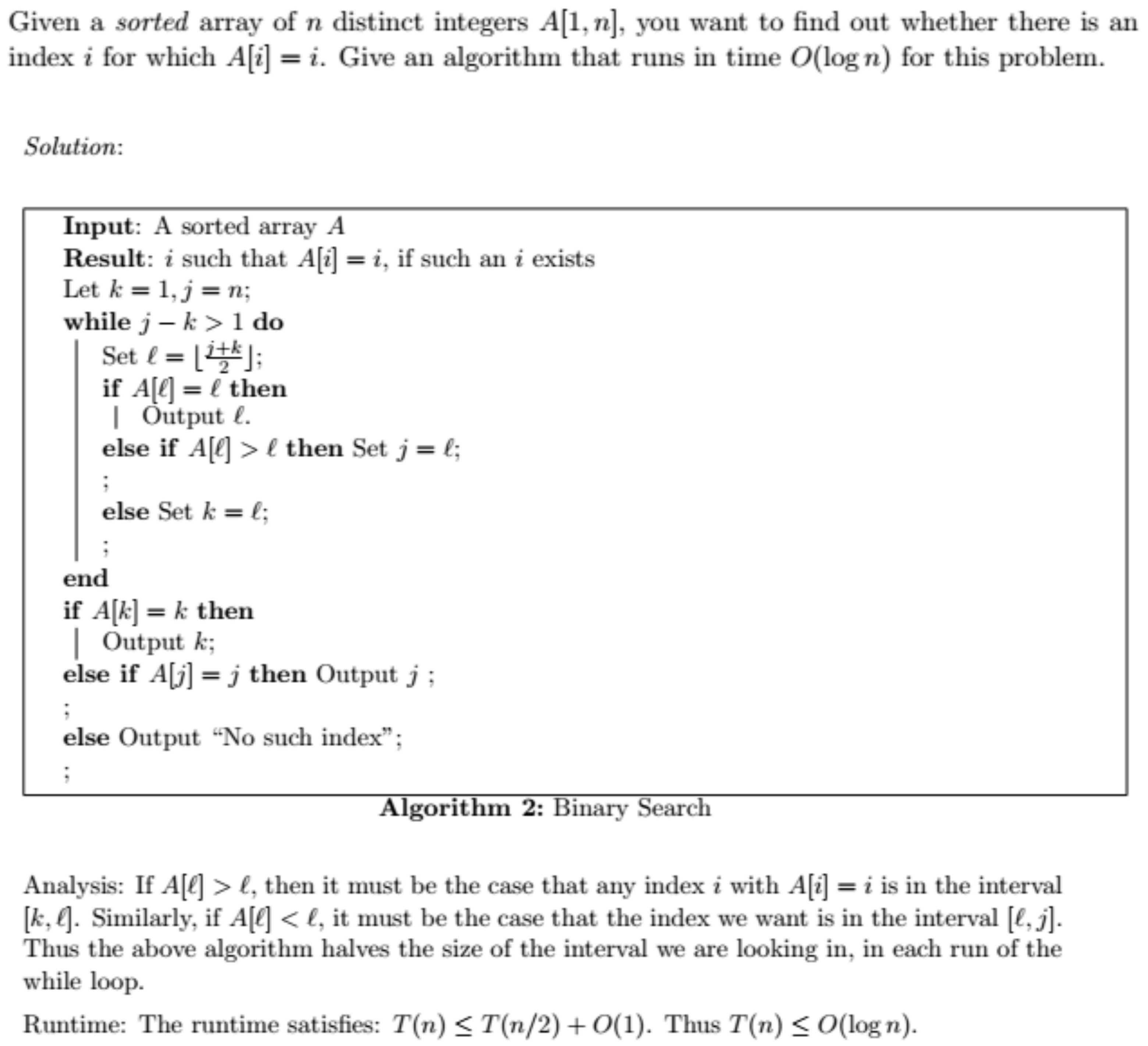
- d) $T(n) = n-1$
  $T(n) = 2T(n/2) + 1$

  Both have some number of additions. However the brute force algorithm will be faster because it has iteration but the other uses recursion.

Problems and Solutions

Given a *sorted* array of n distinct integers $A[1, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give an algorithm that runs in time $O(\log n)$ for this problem.

Problems and Solutions



Given a *sorted* array of n distinct integers $A[1,n]$, you want to find out whether there is an index i for which $A[i] = i$. Give an algorithm that runs in time $O(\log n)$ for this problem.

Solution:

```
Input: A sorted array $A$
Result: $i$ such that $A[i] = i$, if such an $i$ exists
Let $k = 1, j = n$;
while $j - k > 1$ do
    Set $\ell = \lfloor \frac{j+k}{2} \rfloor$;
    if $A[\ell] = \ell$ then
        Output $\ell$.
    else if $A[\ell] > \ell$ then Set $j = \ell$;
    ;
    else Set $k = \ell$;
    ;
end
if $A[k] = k$ then
    Output $k$;
else if $A[j] = j$ then Output $j$ ;
;
else Output "No such index";
;
```

Algorithm 2: Binary Search

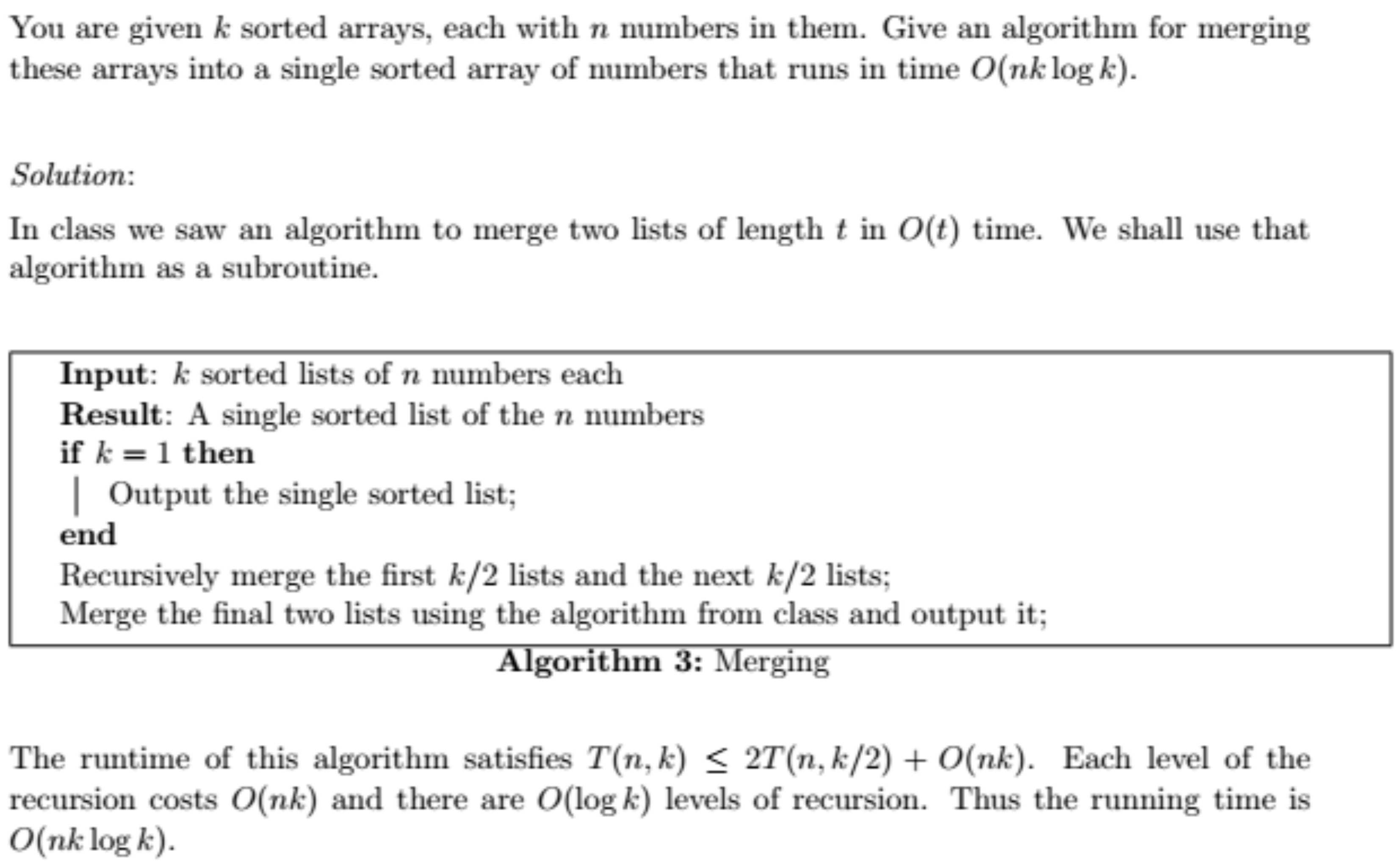
Analysis: If $A[\ell] > \ell$, then it must be the case that any index i with $A[i] = i$ is in the interval $[k, \ell]$. Similarly, if $A[\ell] < \ell$, it must be the case that the index we want is in the interval $[\ell, j]$. Thus the above algorithm halves the size of the interval we are looking in, in each run of the while loop.

Runtime: The runtime satisfies: $T(n) \leq T(n/2) + O(1)$. Thus $T(n) \leq O(\log n)$.

Problems and Solutions

You are given k sorted arrays, each with n numbers in them. Give an algorithm for merging these arrays into a single sorted array of numbers that runs in time $O(nk \log k)$.

Problems and Solutions



You are given k sorted arrays, each with n numbers in them. Give an algorithm for merging these arrays into a single sorted array of numbers that runs in time $O(nk \log k)$.

Solution:

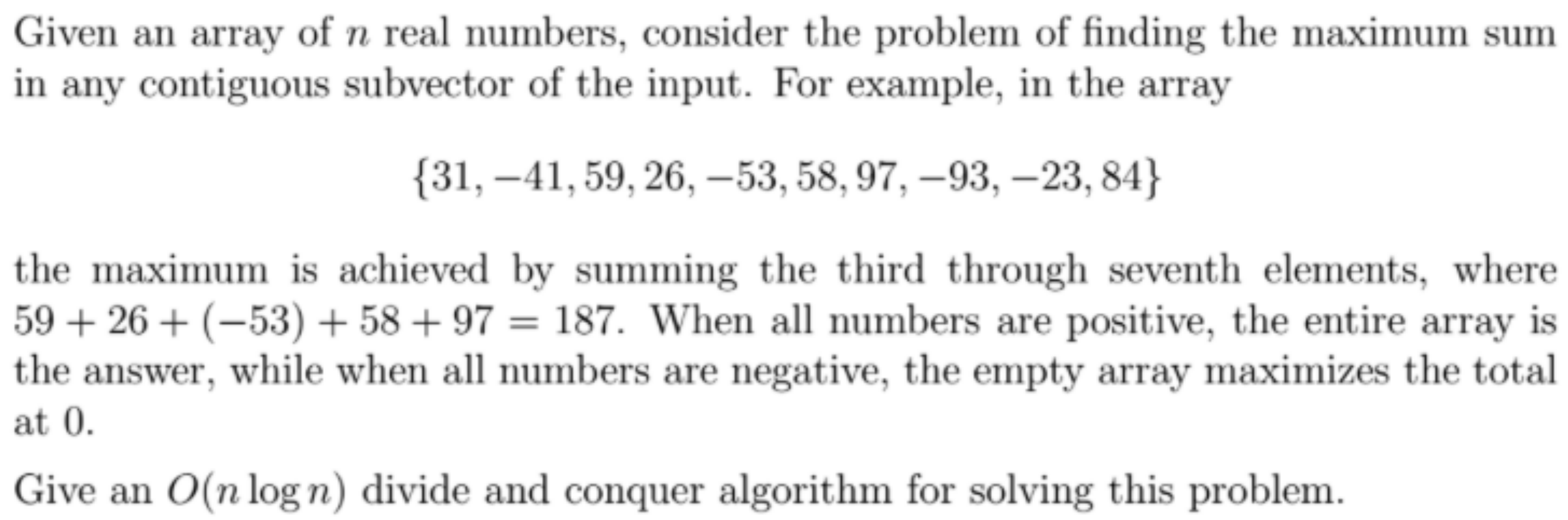
In class we saw an algorithm to merge two lists of length t in $O(t)$ time. We shall use that algorithm as a subroutine.

```
Input: $k$ sorted lists of $n$ numbers each
Result: A single sorted list of the $n$ numbers
if $k = 1$ then
    Output the single sorted list;
end
Recursively merge the first $k/2$ lists and the next $k/2$ lists;
Merge the final two lists using the algorithm from class and output it;
```

Algorithm 3: Merging

The runtime of this algorithm satisfies $T(n, k) \leq 2T(n, k/2) + O(nk)$. Each level of the recursion costs $O(nk)$ and there are $O(\log k)$ levels of recursion. Thus the running time is $O(nk \log k)$.

Problems and Solutions



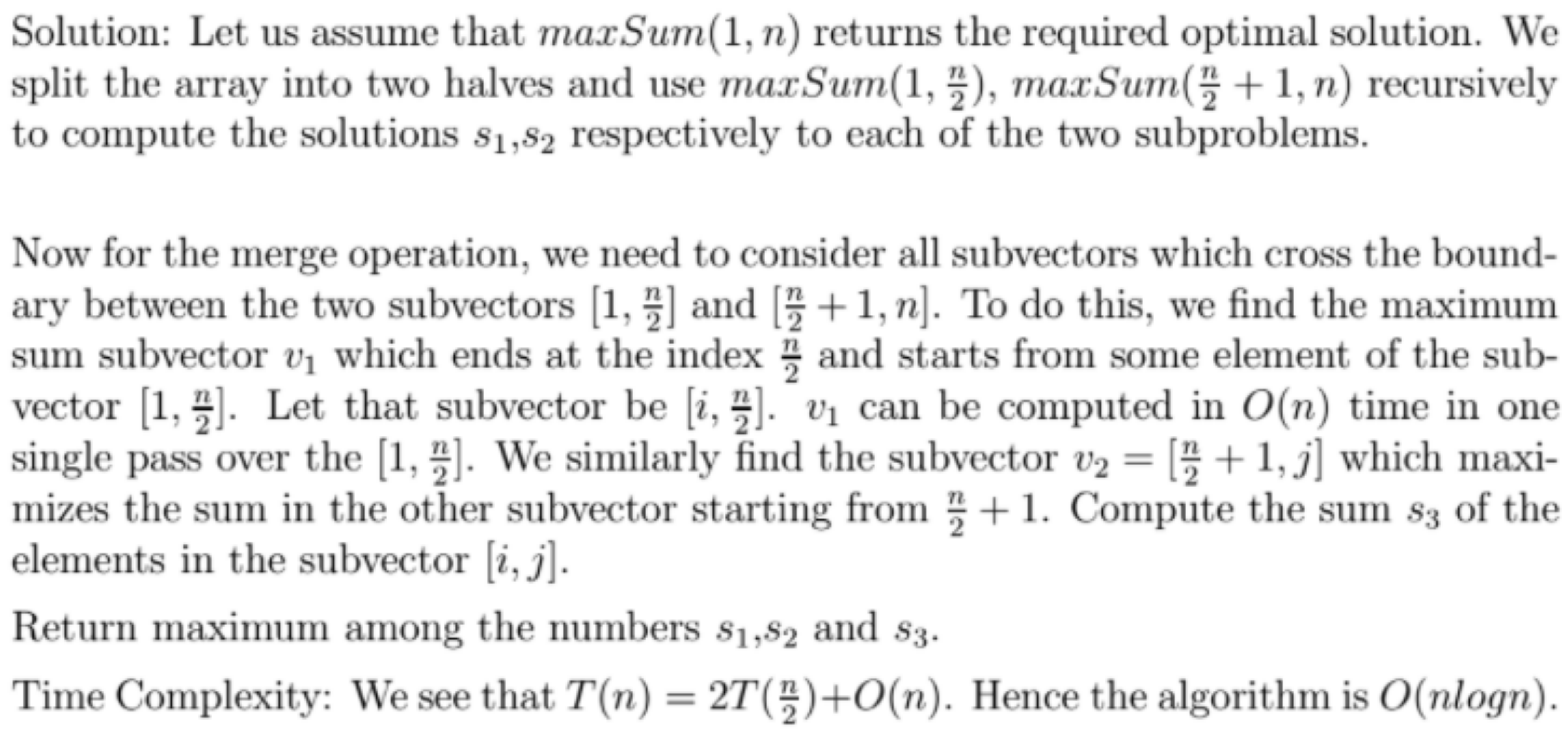
Given an array of n real numbers, consider the problem of finding the maximum sum in any contiguous subvector of the input. For example, in the array

$$\{31, -41, 59, 26, -53, 58, 97, -93, -23, 84\}$$

the maximum is achieved by summing the third through seventh elements, where $59 + 26 + (-53) + 58 + 97 = 187$. When all numbers are positive, the entire array is the answer, while when all numbers are negative, the empty array maximizes the total at 0.

Give an $O(n \log n)$ divide and conquer algorithm for solving this problem.

Problems and Solutions



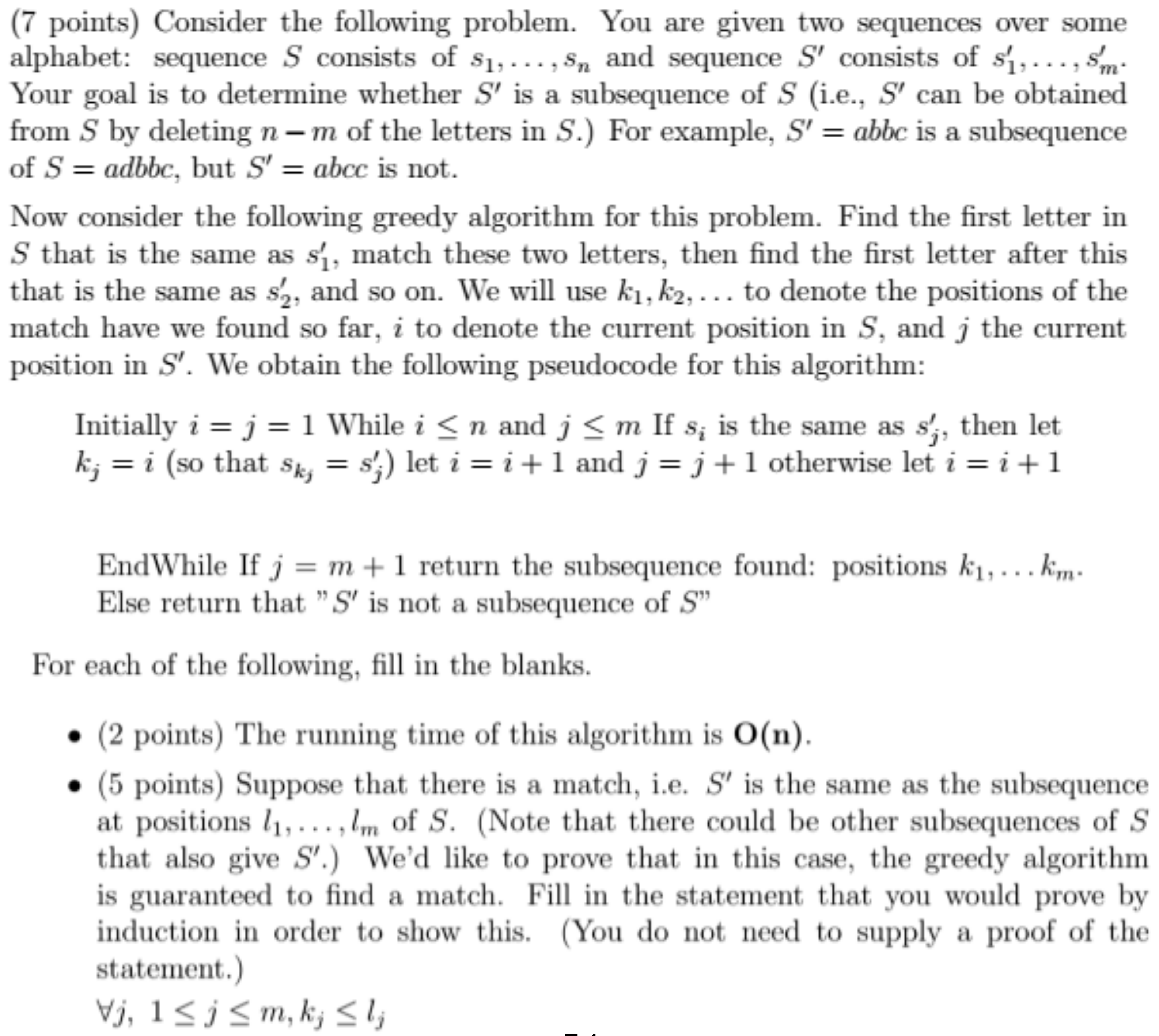
Solution: Let us assume that $maxSum(1, n)$ returns the required optimal solution. We split the array into two halves and use $maxSum(1, \frac{n}{2})$, $maxSum(\frac{n}{2} + 1, n)$ recursively to compute the solutions s_1,s_2 respectively to each of the two subproblems.

Now for the merge operation, we need to consider all subvectors which cross the boundary between the two subvectors $[1, \frac{n}{2}]$ and $[\frac{n}{2} + 1, n]$. To do this, we find the maximum sum subvector v_1 which ends at the index $\frac{n}{2}$ and starts from some element of the subvector $[1, \frac{n}{2}]$. Let that subvector be $[i, \frac{n}{2}]$. v_1 can be computed in $O(n)$ time in one single pass over the $[1, \frac{n}{2}]$. We similarly find the subvector $v_2 = [\frac{n}{2} + 1, j]$ which maximizes the sum in the other subvector starting from $\frac{n}{2} + 1$. Compute the sum s_3 of the elements in the subvector $[i, j]$.

Return maximum among the numbers s_1,s_2 and s_3.

Time Complexity: We see that $T(n) = 2T(\frac{n}{2})+O(n)$. Hence the algorithm is $O(nlogn)$.

Problems and Solutions



(7 points) Consider the following problem. You are given two sequences over some alphabet: sequence S consists of $s_1, \ldots, s_n$ and sequence S' consists of $s'_1, \ldots, s'_m$. Your goal is to determine whether S' is a subsequence of S (i.e., S' can be obtained from S by deleting $n - m$ of the letters in S.) For example, $S' = abbc$ is a subsequence of $S = adbbc$, but $S' = abcc$ is not.

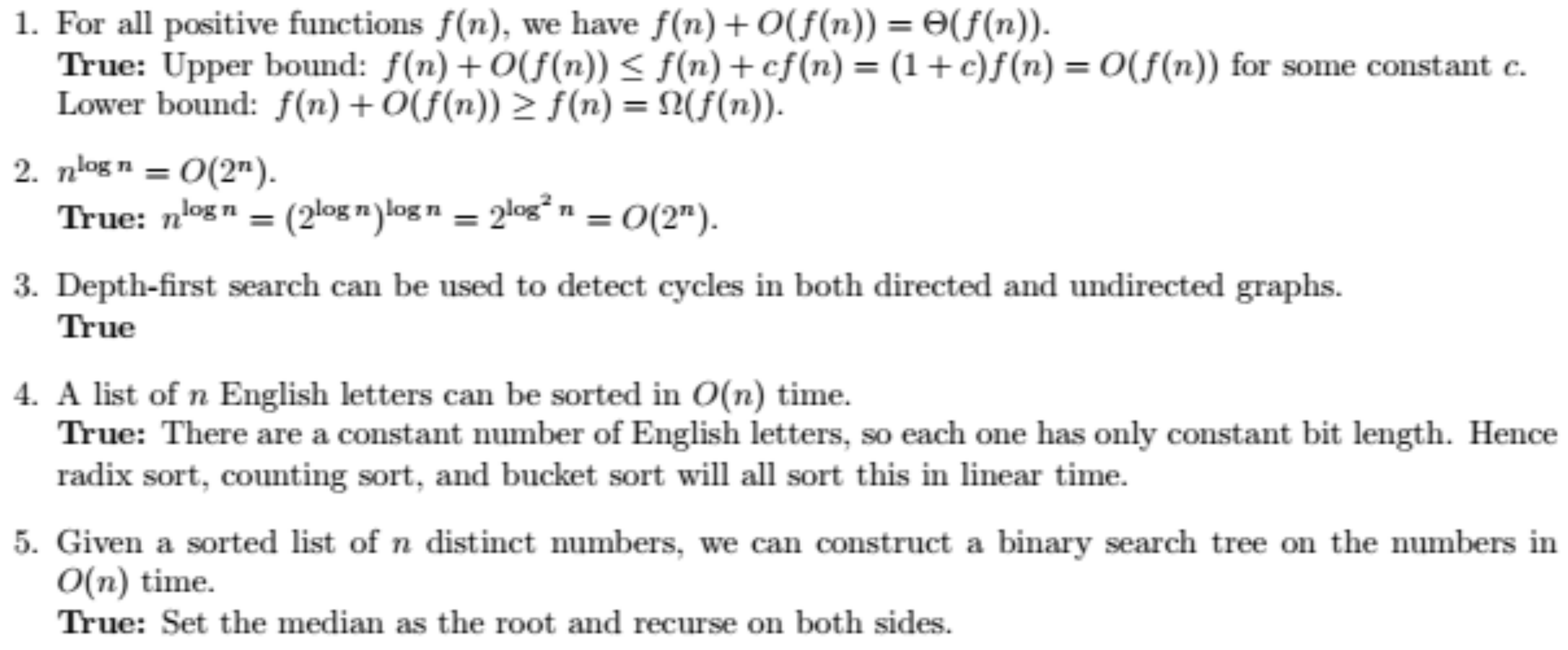
Now consider the following greedy algorithm for this problem. Find the first letter in S that is the same as s'_1, match these two letters, then find the first letter after this that is the same as s'_2, and so on. We will use $k_1, k_2, \ldots$ to denote the positions of the match have we found so far, i to denote the current position in S, and j the current position in S'. We obtain the following pseudocode for this algorithm:

```
Initially $i = j = 1$
While $i \leq n$ and $j \leq m$
    If $s_i$ is the same as $s'_j$, then let $k_j = i$ (so that $s_{k_j} = s'_j$) let $i = i + 1$ and $j = j + 1$
    otherwise let $i = i + 1$
EndWhile
If $j = m + 1$ return the subsequence found: positions $k_1, \ldots k_m$.
Else return that "$S'$ is not a subsequence of $S$"
```

For each of the following, fill in the blanks.

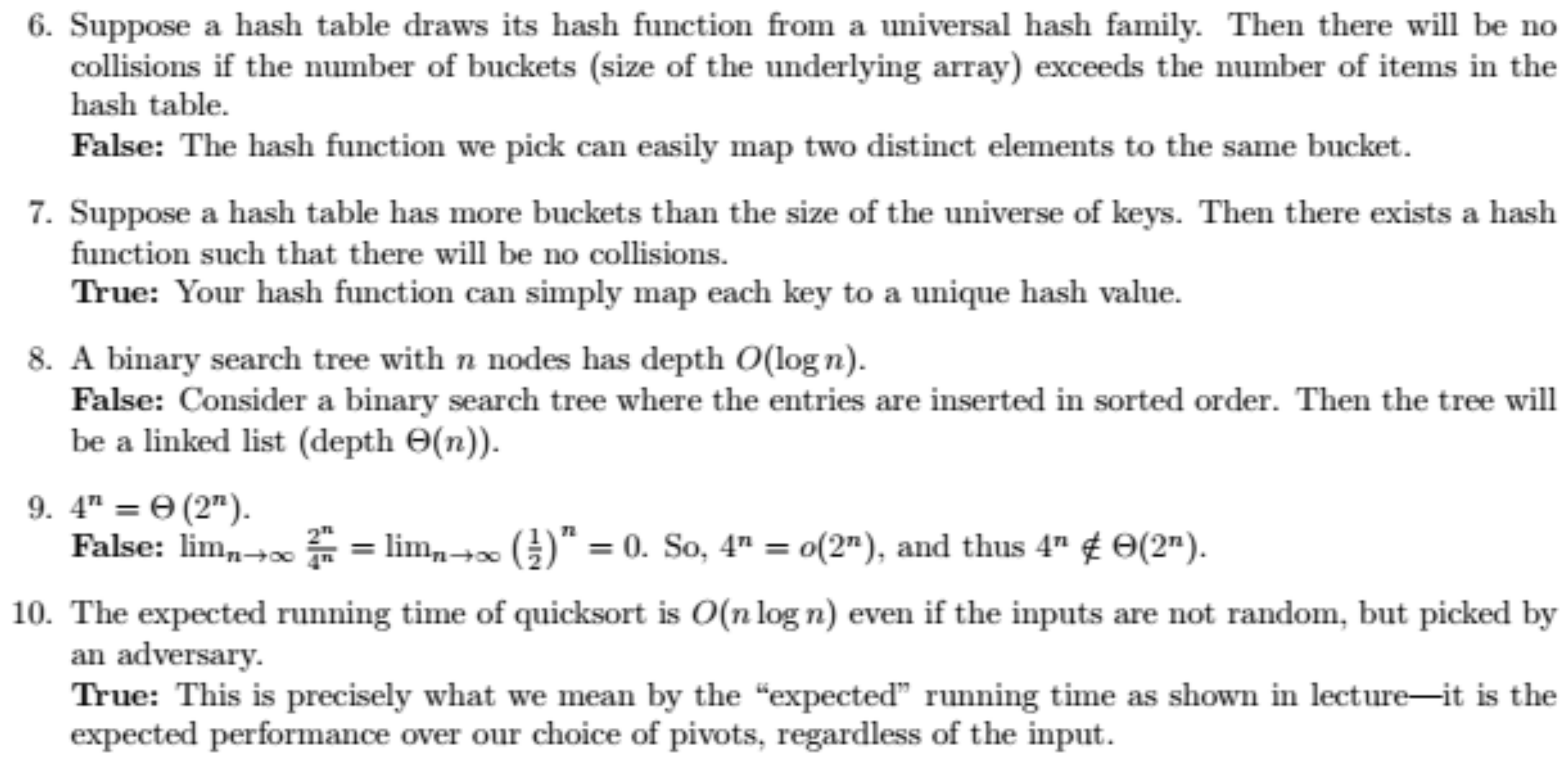
- (2 points) The running time of this algorithm is $\mathbf{O(n)}$.
- (5 points) Suppose that there is a match, i.e. S' is the same as the subsequence at positions $l_1, \ldots, l_m$ of S. (Note that there could be other subsequences of S that also give S'.) We'd like to prove that in this case, the greedy algorithm is guaranteed to find a match. Fill in the statement that you would prove by induction in order to show this. (You do not need to supply a proof of the statement.)
  $\forall j,\ 1 \leq j \leq m, k_j \leq l_j$

Problems and Solutions



1. For all positive functions $f(n)$, we have $f(n) + O(f(n)) = \Theta(f(n))$.
   True: Upper bound: $f(n) + O(f(n)) \leq f(n) + cf(n) = (1+c)f(n) = O(f(n))$ for some constant c.
   Lower bound: $f(n) + O(f(n)) \geq f(n) = \Omega(f(n))$.
2. $n^{\log n} = O(2^n)$.
   True: $n^{\log n} = (2^{\log n})^{\log n} = 2^{\log^2 n} = O(2^n)$.
3. Depth-first search can be used to detect cycles in both directed and undirected graphs.
   True
4. A list of n English letters can be sorted in $O(n)$ time.
   True: There are a constant number of English letters, so each one has only constant bit length. Hence radix sort, counting sort, and bucket sort will all sort this in linear time.
5. Given a sorted list of n distinct numbers, we can construct a binary search tree on the numbers in $O(n)$ time.
   True: Set the median as the root and recurse on both sides.

Problems and Solutions



6. Suppose a hash table draws its hash function from a universal hash family. Then there will be no collisions if the number of buckets (size of the underlying array) exceeds the number of items in the hash table.
   False: The hash function we pick can easily map two distinct elements to the same bucket.

7. Suppose a hash table has more buckets than the size of the universe of keys. Then there exists a hash function such that there will be no collisions.
   True: Your hash function can simply map each key to a unique hash value.

8. A binary search tree with n nodes has depth $O(\log n)$.
   False: Consider a binary search tree where the entries are inserted in sorted order. Then the tree will be a linked list (depth $\Theta(n)$).

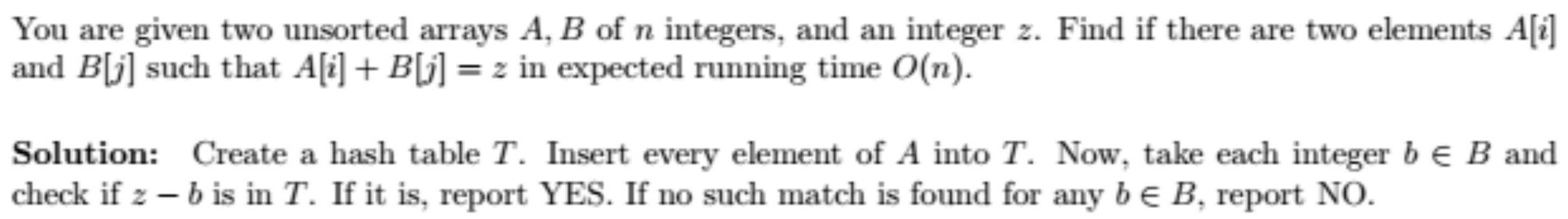
9. $4^n = \Theta\left(2^n\right)$.
   False: $\lim_{n\to\infty} \frac{2^n}{4^n} = \lim_{n\to\infty} \left(\frac{1}{2}\right)^n = 0$. So, $4^n = o(2^n)$, and thus $4^n \notin \Theta(2^n)$.

10. The expected running time of quicksort is $O(n \log n)$ even if the inputs are not random, but picked by an adversary.
    True: This is precisely what we mean by the "expected" running time as shown in lecture—it is the expected performance over our choice of pivots, regardless of the input.

Problems and Solutions

You are given two unsorted arrays A, B of n integers, and an integer z. Find if there are two elements $A[i]$ and $B[j]$ such that $A[i] + B[j] = z$ in expected running time $O(n)$.

Problems and Solutions



You are given two unsorted arrays A, B of n integers, and an integer z. Find if there are two elements $A[i]$ and $B[j]$ such that $A[i] + B[j] = z$ in expected running time $O(n)$.

Solution: Create a hash table T. Insert every element of A into T. Now, take each integer $b \in B$ and check if $z - b$ is in T. If it is, report YES. If no such match is found for any $b \in B$, report NO.