CS 395 – Analysis of Algorithms

Chapter 2 – Fundamentals of the Analysis of Algorithm Efficiency

Read 2.3 & 2.4

- **Mathematical analysis of non-recursive algorithms**
  - General plan for analysis
  - Example: max element problem
  - Example: element uniqueness problem
  - Example: matrix multiplication
  - Example: binary digits
- **Mathematical analysis of recursive algorithms**
  - General plan for analysis
- **Weekly assignment 5: Tower of Hanoi**

To understand recursion you must first understand recursion

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

2

Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+ \cdots +1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

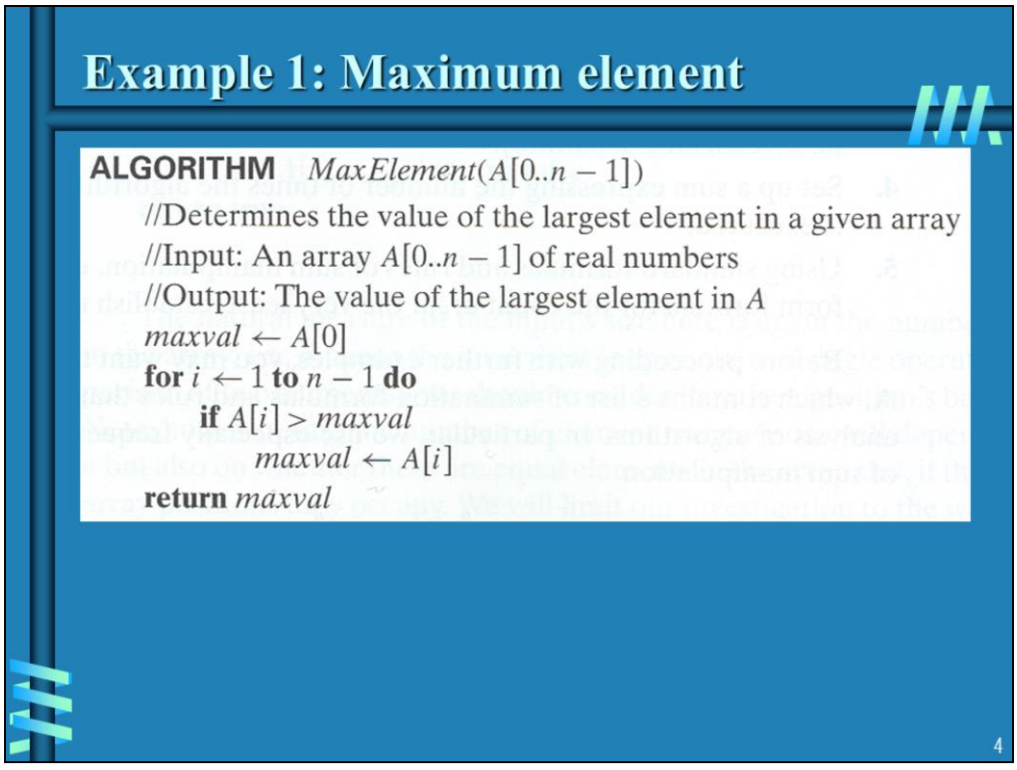
$\Sigma_{1 \le i \le n} i = 1+2+ \cdots +n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+ \cdots +n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i \qquad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$



Maxval <- A[0]

If you do not have –infinity a good strategy is to set it to a possible value to start with.

Example 1: Maximum element

- Input Size: number of elements in the array, *n*
- Most often executed operations are in the *loop*
  - comparison
  - assignment
- Basic operation: comparison
- Formula expressing operation count as function of size *n*

$$C(n) = \sum_{i=1}^{n-1} 1.$$

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

5

Is there a best/worst input for this algorithm? Nope.

Example 2: Element uniqueness problem

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

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//         and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Think it through:
Could you think of another way to do this faster?

6

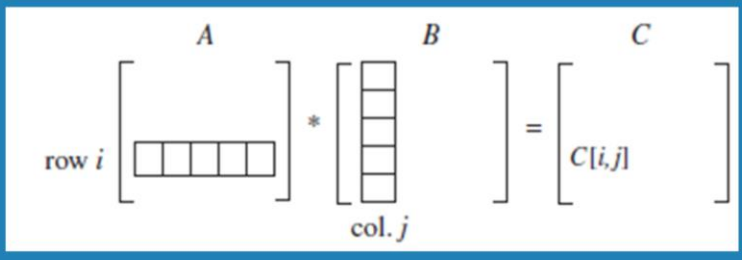
Sort the array O(nlog(n))

Then compare each element with it neighbor O(n)

Example 2: Element uniqueness problem

- Input Size: number of elements in the array, *n*
- Most often executed operation is in the innermost *loop*
  - comparison
- Basic operation: comparison
- Investigate only worst-case scenario(s)
  - No equal element
  - Only the last two elements are equal
- Formula expressing operation count as function of size *n*

Example 3: Matrix multiplication



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \leq i, j \leq n-1$.

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Example 3: Matrix multiplication

- Input Size: matrix order n
- Most often executed operations in the innermost *loop*
  - addition
  - multiplication
- Basic operation: multiplication
- worst-case vs best-case scenarios
- Formula expressing operation count as function of size n

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Example 4: Counting binary digits

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

Example 4: Counting binary digits

- Input Size: the decimal value of the input *n*
- Most often executed operations is the *comparison*
  - But not within the loop
- Basic operation: *comparison*
- How many times loop is executed?
  - $log_2 n$
  - Because *n* is cut in half every time
  - Exact formula for the number of times the comparison n > 1 is executed = $\lfloor \log_2 n \rfloor + 1$

Example 5: The Tower of Hanoi Puzzle

- *n* disks of different sizes
- 3 pegs (A, B, C)
- Goal: move all disks from peg A to peg C using peg B as auxiliary
- Restriction: Can't have a larger disk on top of a smaller one at any time

Let's see what it is:

https://www.youtube.com/watch?v=5Wn4EboLrMM

https://www.mathsisfun.com/games/towerofhanoi.html

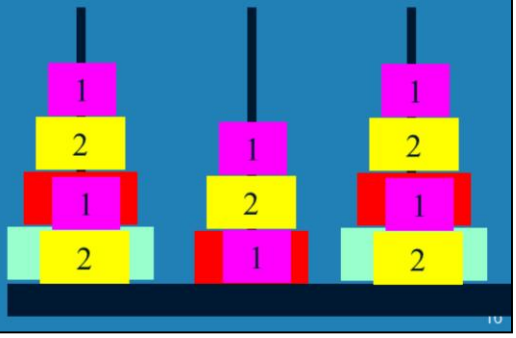
14

Tower of Hanoi

- For i in 1 to 2^n
  - If least significant bit is 1
    - If n is even
      - Move smallest disk right
    - Else
      - Move smallest disk left
  - Else
    - Move disk of least significant bit to the tower it can move to.

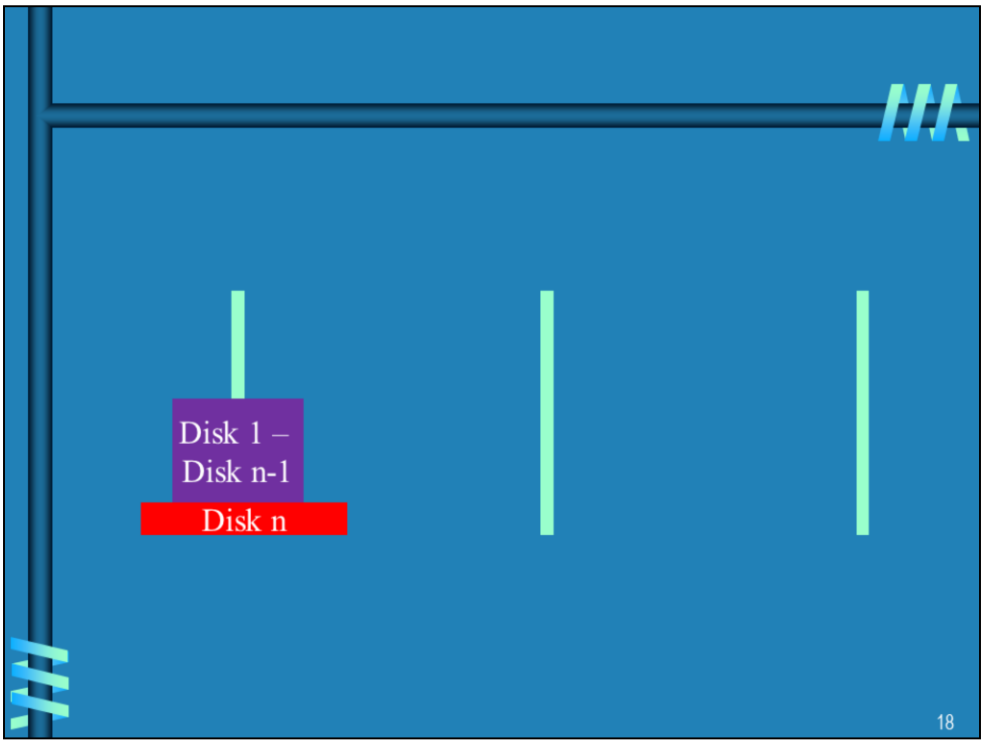
Tower of Hanoi: Example n=4

- 0001 - Move disk 1 right
- 0010 – Move 2 to C
- 0011 - Move disk 1 right
- 0100 - Move 3 to B
- 0101 - Move disk 1 right
- 0110 - Move 2 to B
- 0111 - Move disk 1 right
- 1000 – Move 4 to C
- 1001 - Move disk 1 right
- 1010 – Move 2 to A
- 1011 - Move disk 1 right
- 1100 – Move 3 to C
- 1101 - Move disk 1 right
- 1110 – Move 2 to C
- 1111 - Move disk 1 right



Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic operation is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic operation is executed.
- Solve the recurrence or, at the very least, establish its solution's order of growth (by backward substitutions or another method).


Disk 1 –
Disk n-1
Disk n
18

Example 2: The Tower of Hanoi Puzzle

```
ALGORITHM TowerOfHanoi(nDisks, fromPeg, toPeg, withPeg):
  // Input: number of disks, and peg names

  if nDisks = 1
        print("moving disk from", fromPeg, "to", toPeg)
  else if nDisks > 1
        TowerOfHanoi(nDisks -1, fromPeg, withPeg, toPeg)
        print("moving disk from", fromPeg, "to", toPeg)
        TowerOfHanoi(nDisks -1, withPeg, toPeg, fromPeg)
  return
```

Example 2: The Tower of Hanoi Puzzle

- Input size: n
- Basic Operation: move disk
- Recurrence relation:

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1$$
$$M(1) = 1$$

=>

$$M(n) = 2M(n-1) + 1,$$
$$M(1) = 1$$

Solving recurrence for number of moves

Use *method of backward substitutions*

$$
\begin{aligned}
M(n) &= 2M(n-1)+1 && \text{sub. } M(n-1)=2M(n-2)+1\\
&= 2[2M(n-2)+1]+1 = 2^2M(n-2)+2+1 && \text{sub. } M(n-2)=2M(n-3)+1\\
&= 2^2[2M(n-3)+1]+2+1 = 2^3M(n-3)+2^2+2+1.
\end{aligned}
$$

Generalize:

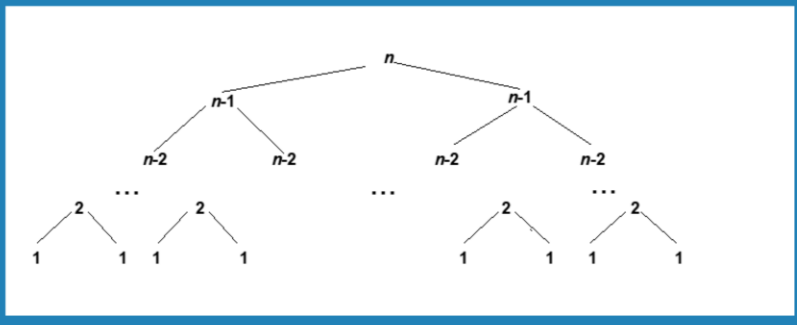
$$M(n) = 2^iM(n-i)+2^{i-1}+2^{i-2}+\cdots+2+1 = 2^iM(n-i)+2^i-1.$$

$$
\begin{aligned}
M(n) &= 2^{n-1}M(n-(n-1))+2^{n-1}-1\\
&= 2^{n-1}M(1)+2^{n-1}-1 = 2^{n-1}+2^{n-1}-1 = 2^n-1.
\end{aligned}
$$

Exponential algorithm.

Tree of calls for the Tower of Hanoi Puzzle

Call Graph:



Number of nodes in the tree = number of calls

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$

22