CSCI 6000 – Data Structures and Analysis of Algorithms

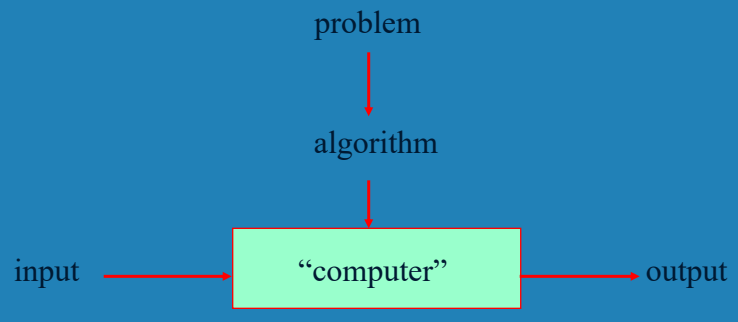
Study Guide for Midterm Exam-1

AUM Computer Science
Fall 2024
Dr. Olcay Kursun

1

What is an algorithm?

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



2

2

Euclid's Algorithm

Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

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

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

3

Two descriptions of Euclid's algorithm

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value fo the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

4

Exercise 1

- In this exercise, you'll implement two algorithms to compute the GCD(m, n) of two numbers—Euclid's recursive GCD (efficient) and a brute force GCD (inefficient). We'll use randomly generated inputs to measure their runtimes, then plot and compare the performance based on input size log(m)+log(n). How do these algorithms scale as the input size grows, and why is Euclid's algorithm faster? Would iterative implementation of GCD run even faster?
- Answer: https://github.com/olcaykursun/Algorithms/blob/main/Fall2024/GCD_practice.ipynb

5

Other methods for computing gcd(*m*,*n*)

Consecutive integer checking algorithm

Step 1 Assign the value of min{*m*,*n*} to *t*

Step 2 Divide *m* by *t*. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide *n* by *t*. If the remainder is 0, return *t* and stop; otherwise, go to Step 4

Step 4 Decrease *t* by 1 and go to Step 2

6

Other methods for gcd(*m*,*n*) [cont.]

Middle-school procedure

Step 1 **Find the prime factorization of *m***

Step 2 **Find the prime factorization of *n***

Step 3 **Find all the common prime factors**

Step 4 **Compute the product of all the common prime factors and return it as gcd(*m*,*n*)**

Is this an algorithm?

7

Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$

for $p \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**

  if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

    $j \leftarrow p * p$

    while $j \leq n$ **do**

      $A[j] \leftarrow 0$ //mark element as eliminated

      $j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

8

Exercise 2

- Apply the Sieve of Eratosthenes upto 30. What is the value of p and j when 12 is first eliminated (labeled as False, non-prime)? How many times is it eliminated (marked as False) in total?
- Answer: You can tell from the pseudocode that p=3 and j=12 when 12 is first eliminated. We can modify the code of Sieve (also used in the solution of Exercise 3) to print out the values of j and p. The number 12 is marked False twice.
- https://github.com/olcaykursun/Algorithms/blob/main/Fall2024/print_prime_sieve_details.ipynb

9

Exercise 3

- Change the Middle-school procedure outlined in the previous slide to an algorithm?
- Answer: https://github.com/olcaykursun/Algorithms/blob/main/Fall2024/middle_school_gcd_practice.ipynb

10

Exercise 4: Why study algorithms?

- Theoretical importance
  - the core of computer science
- Practical importance
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

11

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

12

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency
- Does there exist a better algorithm?
  - lower bounds
  - optimality

13

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

14

Analysis of algorithms

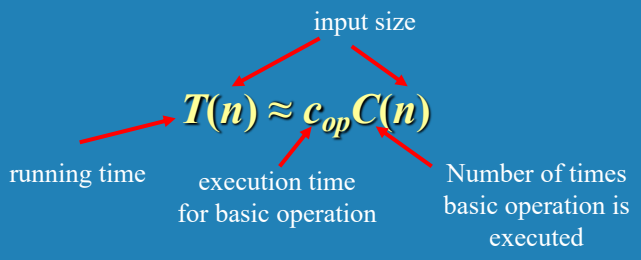
- **Issues:**
  - correctness
  - time efficiency
  - space efficiency
  - optimality
- **Approaches:**
  - theoretical analysis
  - empirical analysis

15

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- ***Basic operation*: the operation that contributes most towards the running time of the algorithm**



16

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

17

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)
  or
  Count actual number of basic operation's executions
- Analyze the empirical data

18

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- **Worst case:** $C_{worst}(n)$ **– maximum over inputs of size** n
- **Best case:** $C_{best}(n)$ **– minimum over inputs of size** n
- **Average case:** $C_{avg}(n)$ **– "average" over inputs of size** n
  - **Number of times the basic operation will be executed on typical input**
  - **NOT the average of worst and best case**
  - **Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs**

19

Exercise 5: Sequential search complexity?

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//         or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- **Worst case**
- **Best case**
- **Average case**

20

Types of formulas for basic operation's count

- **Exact formula**
  e.g., $C(n) = n(n-1)/2$
- **Formula indicating order of growth with specific multiplicative constant**
  e.g., $C(n) \approx 0.5\, n^2$
- **Formula indicating order of growth with unknown multiplicative constant**
  e.g., $C(n) \approx cn^2$

21

Order of growth

- **Most important: Order of growth within a constant multiple as $n \to \infty$**
- **Example:**
  - **How much faster will algorithm run on computer that is twice as fast?**
  - **How much longer does it take to solve problem of double input size?**

22

Values of some important functions as $n \to \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

23

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- **$O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$**
- **$\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$**
- **$\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$**

24

Big-oh

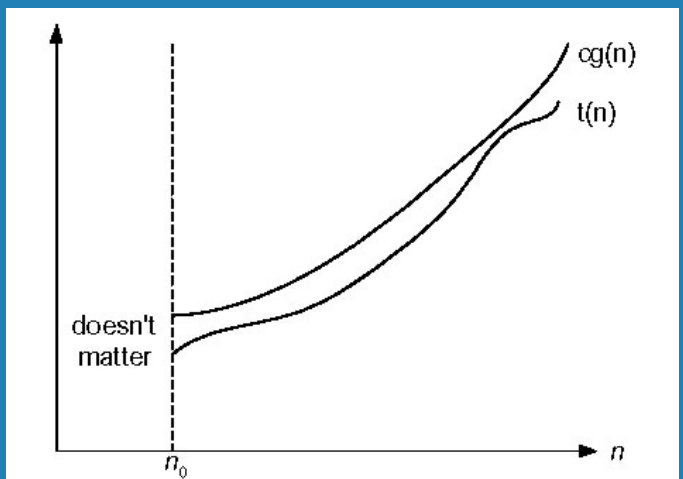


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

25

25

Big-omega

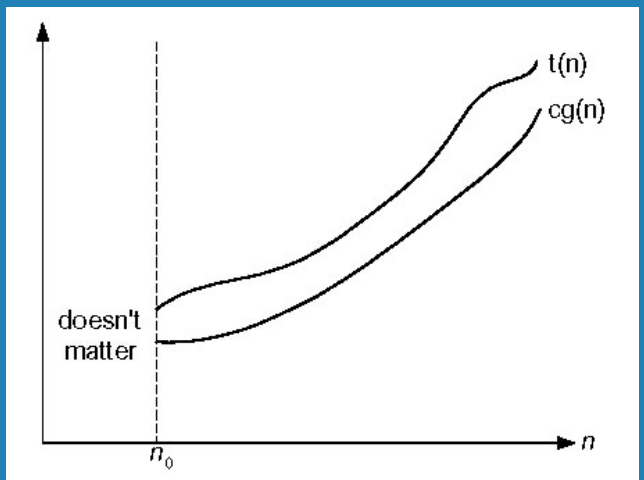


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

26

26

Big-theta

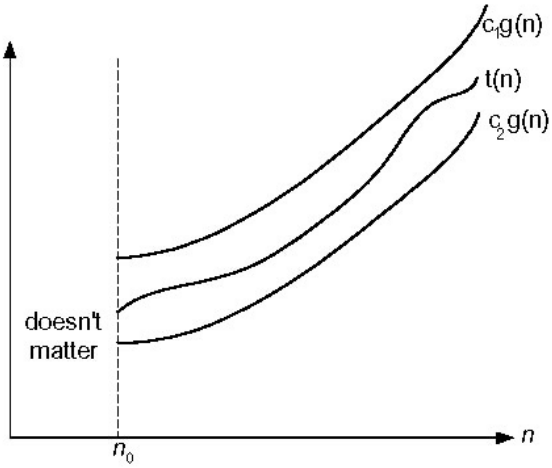


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

27

Establishing order of growth using the definition

Definition: $f(n)$ **is in** $O(g(n))$ **if order of growth of** $f(n) \leq$ **order of growth of** $g(n)$ **(within constant multiple), i.e., there exist positive constant** c **and non-negative integer** n_0 **such that**

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n$ **is** $O(n^2)$
- $5n+20$ **is** $O(n)$

28

Exercise 6

Show the following

$10n$ is $O(n^2)$

$5n+20$ is $O(n)$

$5n^2+2n+1000$ is $O(n^2)$

29

29

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

  $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

30

30

Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is
- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$
- Exponential functions a^n have different orders of growth for different a's
- order $\log n$ < order n^α ($\alpha>0$) < order a^n < order $n!$ < order n^n

31

Basic asymptotic efficiency classes

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n or linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

32

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)

33

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

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

34

Example 1: Maximum element

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

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```


35

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
```


36

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
```


37

Plan for Analysis of Recursive Algorithms

- It cannot be investigated the way the previous (iterative) examples are investigated with the summations for the loops.
- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. 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 op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

38

Recursive evaluation of *n*!

Definition: $n! = 1 \cdot 2 \cdot \ldots \cdot (n\text{-}1) \cdot n$ for $n \geq 1$ and $0! = 1$

Recursive definition of *n*!: $F(n) = F(n\text{-}1) \cdot n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) ∗ n
```

Size:
Basic operation:
Recurrence relation:

Solving the recurrence for M(n) of n!

M(*n*) = M(*n*-1) + 1, M(0) = 0, where M is the number of multiplications (not the time complexity).

You can solve this to obtain a closed-form formula: M(n) = n

So what is the time complexity, T?

When n is given, let m be the size of n, which is log(n). Therefore, $T(m) = n = 2^m$, which is exponential.

Exercise 7: Counting binary digits

ALGORITHM *Binary*(n)

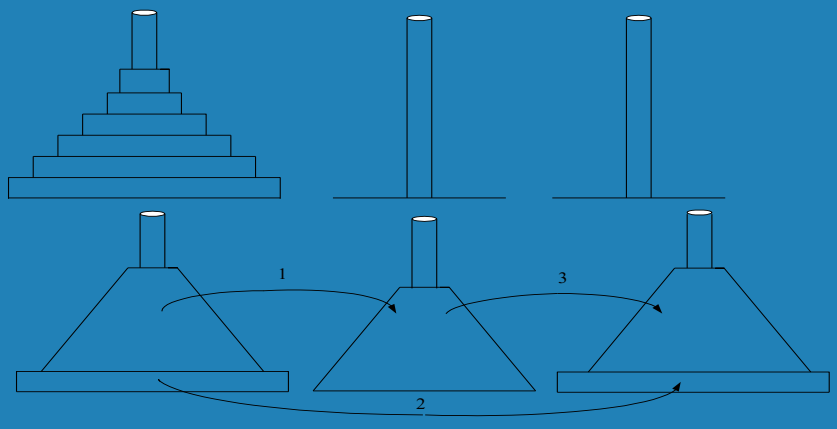
```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

Write and analyze the recurrence to show that D(n) = log(n) where D denotes the number of divisions.

True or False: As the complexity should use the size of the instance, we can say T(m) = m, where m is log(n).

41

Another Example for Analyzing Recurrence Relations: The Tower of Hanoi Puzzle



Recurrence for number of moves:

$M(n) = 2M(n\text{-}1) + 1,\ M(1) = 1$

42

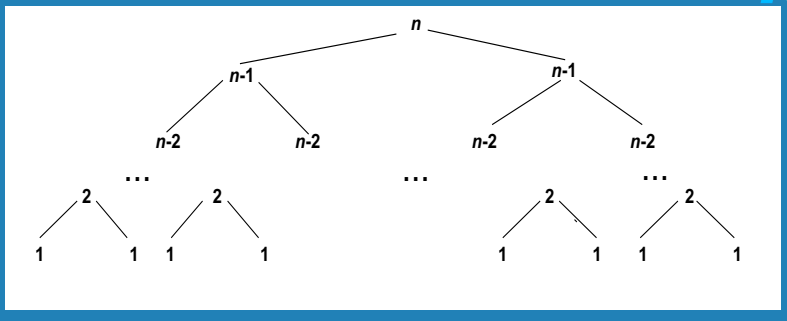
Exercise 8

Solving the recurrence for M(n) of Towers of Hanoi

$M(n) = 2M(n\text{-}1) + 1,\ M(1) = 1$

43

Tree of calls for the Tower of Hanoi Puzzle



44

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for duplicates or a key of a given value in a list

45

Exercise 9

- Answer the following for the solution of Element Uniqueness problem presented on Slide 22:
- What is the basic operation (which line)?
- Using summations, calculate how many times is the basic operation executed.
- Give an exemplary instance for each of the best, worst, and average case runtimes for this solution.

- Answer (sketch): For the summations, we solved something very similar for the closest pair problem. Worst case is when they are all unique, the best case is when the first two items are equal. The average case occurs when the duplicate is found neither at the beginning nor at the very end of the process, for example [1 2 3 4 2]. On average, this leads to roughly half of all possible comparisons being made.

46

Brute-Force Sorting Algorithm

Selection Sort **Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:**

$$A[0] \le \ldots \le A[i-1] \mid A[i], \ldots, A[min], \ldots, A[n-1]$$

in their final positions

Example: 7 3 2 5

47

Analysis of Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

Time efficiency:

Space efficiency:

Stability:

48

Non-Brute-Force Alternative to Selection Sort

- **Insertion Sort (Decrease and Conquer): To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]. Usually implemented bottom up (nonrecursively)**

Example: Sort 6, 4, 1, 8, 5

6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8

- **Time efficiency**

$C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

$C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

$C_{best}(n) = n - 1 \in \Theta(n)$ **(fast on almost sorted arrays and considered the best elementary sorting algorithm)**

Exercise 10

- **Implement selection sort and insertion sort and compare their runtimes on a large array.**

Brute-Force String Matching

- ***pattern***: a string of *m* characters to search for
- ***text***: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until
- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Efficiency:

Examples of Brute-Force String Matching

Pattern: 001011

Text: 10010101101001100101111010

Pattern: happy

Text: It is never too late to have a happy childhood.

53

Exercise 11

- Implement the brute-force solution to the string matching. Run it on the examples provided in the previous slide. Count how many character comparisons are performed for each example.
- What is a good alternative to this brute-force algorithm?
- Answer: Non-Brute-Force Alternative would analyze the pattern and based on the mismatch (performed right-to-left), it would shift the pattern on text more efficiently (instead of one character at a time when comparisons are performed left-to-right).

54

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 indexes of the points for which the distance is the smallest.

55

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//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 $index1, index2$

Efficiency:

How to make it faster?

56

Exercise 12

- Implement a solution to the Closest Pair problem. For the instance of the problem, generate 10 random points and find/print the distance of the closest pair.
- There is a non-brute-force solution for this problem and it is easiest to explain it on a 1D example: To find the closest pair when the n points, A[0..n-1], were located on a number line. Provide an algorithm (pseudo-code) using an effective (n*logn) sort algorithm (lets just call it "sort(A[0..n-1])".

57

Brute-Force Strengths and Weaknesses

- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)
- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

58

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 (see algorithms in Sec. 5.4)
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

59

Example: 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

60

Exercise 13: Knapsack Problem by Exhaustive Search

Fill the table below that explores/exhausts all possibilities, how many such possibilities exist?

Subset	Total weight	Total value
Answer:		
{}	0	$0
{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

Non-Brute-Force Alternative to Knapsack Prob

We learned (and will learn more) that dynamic programming can be another alternative to brute-force for certain problems.

Write a short program that utilizes an array to solve the Fibonacci numbers problem: Find the nth Fibonacci number. Example: for n=7 your program will produce the array [0, 1, 1, 2, 3, 5, 8, 13] with indices from 0 to 7. That output indicates that the 7th Fibonacci number is 13.

Write a short program that utilizes an array to solve the minimum coin exchange problem. Given the denomination [1, 3, 4] and n=6, your program will output the following array: [0, 1, 2, 1, 1, 2, 2] with indices from 0 to 6. That output indicates that 6 can be obtained with the sum of 2 coins.

Non-Brute-Force Alternative to Knapsack

Given n items of

integer weights: w_1 w_2 … w_n

values: v_1 v_2 … v_n

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \le W$).

Let $V[i,j]$ be optimal value of such instance. Then

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j - w_i \ge 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

63

63

Exercise 14: Knapsack Problem (efficient solution)

Fill the table: Knapsack of capacity $W = 5$

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

	capacity j					
	0	1	2	3	4	5
0						
$w_1 = 2, v_1 = 12$ 1						
$w_2 = 1, v_2 = 10$ 2						
$w_3 = 3, v_3 = 20$ 3						
$w_4 = 2, v_4 = 15$ 4						?

64

64

Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search or its variation is the only known way to get exact solution

65

65