

## Topic 4: Brute-force algorithms

1. Numeric computations
2. Vector traversal
3. Sorting
4. Points in  $n$ -space
5. Exhaustive search

*Reading:* Sec. 11.1; Levitin Ch 2

## Brute force

- *Definition:* “a straightforward approach, usually based directly on the problem’s statement...” (Levitin, p. 97)
- *Advantages:*
  - applicable to a wide range of problems
  - simple to design;
  - may be useful to solve small problem instances
- *Disadvantage:* inefficient

## 1. Numeric computations

### Multiplication and exponentiation

$Product(a, b) =$

$$\begin{cases} a & \text{if } b = 1 \\ a + Product(a, b-1) & \text{otherwise} \end{cases}$$

$Pow(a, b) =$

$$\begin{cases} 1 & \text{if } b = 0 \\ a \times Pow(a, b-1) & \text{otherwise} \end{cases}$$

### Decrease by one

#### Factorial (n)

```

if n ≤ 1
    return 1
else
    return n × Factorial(n - 1)

```

#### Fibonacci (n)

```

if n > 0
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)

```

## Matrix multiplication

- Given two 2-dimensional matrices, their *product* is a matrix whose cells are each the sum of the products of several cells chosen from both matrices
- Given square matrices  $A$  and  $B$ , let  $C$  be the product;

$$C[i, j] = A[i, 0] B[0, j] + \dots$$

$$A[i, k] B[k, j] + \dots$$

$$A[i, n-1] B[n-1, j] \text{ for } i, j \leq n - 1$$

## Matrix multiplication algorithm

**Matmult (M[1..hM, 1..vM], P [1..hP, 1..vP])**

> Pre:  $vM = hP$

for  $i \leftarrow 1$  to  $vM$

  for  $j \leftarrow 1$  to  $hM$

$Y[i, j] \leftarrow 0$

    for  $k \leftarrow 1$  to  $hM$

$Y[i, j] \leftarrow Y[i, j] + M[j] \dots$

return  $Y$

## 2. Vector traversal

### Linear search

#### Search(A, x)

```
for  $i \leftarrow 1$  to  $|A|$ 
  if  $[i] = x$  return true
return false
```

#### Search(A, x) =

$$\begin{cases} \textit{false} & \text{if } |A| = 0 \\ \textit{true} & \text{if } A[1] = x \\ \textit{Search}(A[2..|A|], x) & \text{otherwise} \end{cases}$$

### Linear-search recurrence

- To tell whether array  $A$  contains value  $key$  in range  $A[first\dots last]$ :
- $\textit{Search}(A, first, last, key) =$ 

$$\begin{cases} \textit{false} & \text{if } first > last \\ \textit{true} & \text{if } A[first] = key \\ \textit{Search}(A, first+1, last, key) & \text{otherwise} \end{cases}$$
- This recursive definition is equivalent to the iterative version

## Brute-force string search

Finds location of first occurrence of *key* in *S*

Search(*S* [1..*n*], *key* [1 .. *m*])

```

for  $i \leftarrow 1$  to  $n - m + 1$  do
   $j \leftarrow 1$ 
  while  $j \leq m \wedge \text{key}[j] = S[i + j]$  do
     $j \leftarrow j + 1$ 
  if  $j > m$  return  $i$ 
Return  $-1$ 

```

Worst-case complexity? \_\_\_\_\_

Average-case? \_\_\_\_\_

## Recurrences for string search

$$\text{Match}(S_1, S_2) = \begin{cases} \text{true} & \text{if } |S_1| = |S_2| = 0 \\ \text{false} & \text{if } S_1[1] \neq S_2[1] \\ \text{Match}(S_1[2..|S_1|], S_2[2..|S_2|]) & \text{otherwise} \end{cases}$$

$$\text{Search}(S_1, S_2) = \begin{cases} \text{true} & \text{if } \text{Match}(S_1, S_2) \\ \text{false} & \text{if } |S_1| < |S_2| \\ \text{Search}(S_1[2..|S_1|], S_2) & \text{otherwise} \end{cases}$$

## Complexity of array insertion

### Insert-ascending (A, x)

```

i ← |A|
while i > 1 and x < A[ i ]
  A[ i ] ← A[ i - 1 ]
  i ← i - 1
A[ i ] ← x
Return A

```

*Inserts x at its proper location in ascending sequence of A's elements*

- Amount of data ( $n$ ) initializes counter
- Loop iterates up to  $n$  times
- $T_{\text{Insert}}(n) = O(n)$

## Recursive array insertion

### Insert-ascending (A, x)

```

If |A| = 0
  return ⟨x⟩
else
  if x < A[1]
    return ⟨x⟩ + A
  else
    return ⟨A [1]⟩ +
      Insert-ascending(A[2 .. |A|], x)

```

**Insert-ascending (A, x) =**

$$\begin{cases} \langle x \rangle & \text{if } |A| = 0 \\ \langle x \rangle + A & x < A[1] \\ \langle A[1] \rangle + \text{Insert-ascending}(A[2 .. |A|], x) & \text{otherwise} \end{cases}$$
  

$$T_{\text{insert-asc}}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ O(1) + T_{\text{insert-asc}}(n - 1) & \text{otherwise} \end{cases}$$

Analysis of Algorithms David Keil 5/08 13

## 3. Sorting

### Insertion sort

```

i ← 1
While i < |A|
  Insert-ascending(A[1 .. i], A[i + 1])
  i ← i + 1
Return A
        
```

1

2	1	6	3	4
---	---	---	---	---

→

1	2	6	3	4
---	---	---	---	---

2

sorted unsorted

sorted unsorted

Complexity analysis: O( ? ) Why?

David Keil Analysis of Algorithms 4. Brute force 8/08 14

## Recursive insertion sort

### Insertion-sort (A)

If  $|A| \leq 1$

return A

Else

return *Insert-ascending*

(*Insertion-sort*(A [1 .. |A| - 1]), A[|A|] )

### Insertion-sort (A) =

$$\begin{cases} A & \text{If } |A| \leq 1 \\ \text{Insert-ascending} \\ \quad (\text{Insertion-sort}(A [1 .. |A| - 1]), A[|A|]) & \\ \text{otherwise} \end{cases}$$

$$T_{ins-sort}(n) =$$

$$\begin{cases} O(1) & \text{if } n \leq 1 \\ O(n) + T_{ins-sort}(n - 1) & \text{otherwise} \end{cases}$$

## Selection Sort, recursive version

***Selection-sort*** ( $A, start, size$ )

If  $size > 1$

1. Find lowest value in  $A[start..size]$
2. Swap it with  $A[start]$
3. ***Selection-sort*** ( $A, start + 1, size$ )

*Time-complexity recurrence:*

$$T_{sel-sort}(n) = \begin{cases} 1 & \text{if } n = 1 \\ O(n) + 5 + T_{sel-sort}(n-1) & \text{if } n > 1 \end{cases}$$

①
②
③

$$T_{sel-sort}(n) = O(n^2)$$

## Solving recurrences: decrease-by-one

- Decrease-by-one algorithms exploit relationship between  $f(n)$  and  $f(n-1)$ , e.g.,  $n! = n(n-1)!$
- These algorithms' time efficiency may be expressed in the form  $T(n) = T(n-1) + f(n)$  where  $f(n)$  expresses time to reduce and extend between smaller and larger instances
- $T(n) = T(n-1) + f(n)$   
 $= T(n-2) + f(n-1) + f(n) \dots$

## 4. Points in $n$ -space

### Brute-force closest-pair

- *Problem:* From a set of all pairs of points on a coordinate graph, select the pair of points that are in closest proximity,
- *Solution:* For all pairs of points, compute distances, find points that generate the minimum of this set

pic

### Brute-force convex hull

- *Problem:* Find smallest convex polygon that contains all of a set of points
- *Solution:* Generate all segments joining pairs of points; select those segments for which all other points are on same side of that segment

pic

## 5. Exhaustive search

- Generate each element of a problem domain, select best-valued one that satisfies constraint
- *Example:* Traveling Salesperson Problem, finding minimum-cost path through all cities on a map. *Solution:* Compare costs of all paths
- *Example:* Knapsack Problem
  - *Problem:* find maximum-valued subset of weighted items under a given total weight.
  - *Solution:* Find total weight of each subset, find subset with maximum value where weight does not exceed maximum weight

## Concepts

brute force  
brute-force closest-pair  
brute-force convex hull  
brute-force knapsack  
brute-force Traveling Salesperson  
Bubble sort  
*Drag-max*  
exhaustive search  
*Insert-ascending*  
insertion sort  
linear search  
matrix multiplication

### References

- A. Levitin. *The Design and Analysis of Algorithms*. Addison Wesley, 2003.  
Chapter 2.
- R. Johnsonbaugh and M. Schaefer.  
*Algorithms*. Pearson Prentice Hall, 2004.  
Sec. 11.1.
- D. Keil classroom work.