

**Topic 3:**  
**Algorithm analysis;**  
**recurrence relations;**  
**brute force**

*Reading: Ch. 2*

David Keil Analysis of Algorithms 3. Recurrences 1/11 1

**Subtopics on analysis and recurrences**

1. Big-O notation
2. Algorithm analysis
3. Recurrence relations and time functions
4. Solving recurrences

**Objectives**

- 3a. Define and use the big-O, theta, and big-omega notations
- 3b. Write recurrences that define the function computed by a simple linear-time algorithm, and its time function

David Keil Analysis of Algorithms 3. Recurrences 1/11 2

## 1. Big-O notation

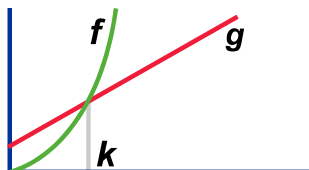
- Quantifies and classifies the growth rates of functions
- May express the time efficiency of an algorithm as a function of amount of data processed
- One algorithm is “*asymptotically faster*” than another if it is faster for all sizes of inputs beyond some constant
- *Examples:*
  - $O(n + 1,000,000) = O(n)$
  - $O(0.2 \lg n) = O(\lg n)$
  - $O(k n^2) = O(n^2)$  for any constant  $k$

## Upper and lower bounds

- Function  $g$  is an *upper bound* on function  $f$  iff  $(\forall x \in \mathbf{N}) g(x) \geq f(x)$
- Suppose
  - $f(x) = x^2$
  - $g(x) = x^3$
  - $h(x) = (x - 1) / 2$
- Then
  - $g$  is an upper bound on  $f$
  - $f$  is a lower bound on  $g$
  - $h$  is a lower bound on  $f$  and  $g$(See graphs of the functions)

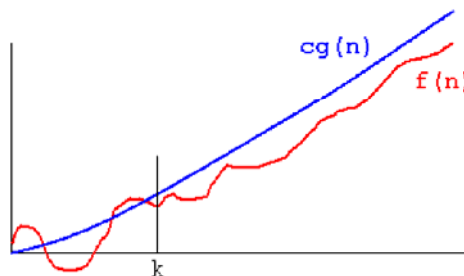
## Asymptotic behavior

- Different time functions  $T$  have graphs that “cross over” each other
- For amounts of data up to  $k$ , time function  $f$  is “better,” so its algorithm is more efficient
- But above  $k$ ,  $g$  shows less time
- We choose the algorithm described by  $g$ , because eventually (asymptotically) its time value  $T$  will be less than  $f$ 's
- ...because  $g$  grows more slowly than  $f$



## Big O notation

- $f \in O(g)$  iff  
 $(\exists c, k) (\forall x \geq k) 0 \leq f(x) \leq cg(x)$
- Hence  $O(g)$  is the set of functions for which  $g$  is an upper bound within a constant factor
- The graph of  $cg(n)$  rises above the graph of  $f(n)$  everywhere for  $n > k$  (never cross again)



## Grouping time functions

- Functions that grow roughly at the same rate are in the same Big-O category
- Constant time:  $O(1)$
- Logarithmic time:  $O(\log_2 n) = O(\lg n)$
- Linear time:  $O(n)$
- Quadratic time:  $O(n^2)$
- Exponential time:  $O(2^n)$
- We abstract and simplify to obtain hardware-independent results:  
 $O(n^2 + 5n) = O(n^2)$ ,  $O(n + \log n) = O(n)$

## Dominance relations on functions

- Our goal is to obtain a hierarchy of sets of time functions to classify algorithm runtimes
- $O(n)$  functions dominate  $O(\lg n)$ ,  $O(n^2)$  dominates  $O(n)$ , etc.
- Non-dominant terms in time functions are discarded
- Constant terms and factors are discarded
- *Example:*  $O(3n^2 + 2n - 100)$  simplifies to  $O(n^2)$

## Big-omega notation ( $\Omega$ )

- Expresses *lower bound* on time for optimistic or best case
- *Example:* A *Find-max* algorithm for arrays is  $\Omega(n)$  because any algorithm must check every array element
- Precise characterization of comparison-based sorting:  $\Omega(n \lg n)$
- A *problem's* complexity may be expressed using  $\Omega$  if the problem is known to have a lower bound on time

## Big-omega and lower bound

- $f \in \Omega(g)$  iff  
 $(\exists c, n_0) (\forall x \geq n_0) 0 \leq cg(x) \leq f(x)$
- [pic] the graph of  $f(n)$  rises above the graph of  $cg(n)$  everywhere for  $n > n_0$  (never cross again)
- $\Omega$  characterizes *problems*; e.g., searching unordered array is  $\Omega(n)$ ; sorting by comparisons is  $\Omega(\lg n)$

## Theta notation ( $\Theta$ )

- $\Theta$  expresses a *tight* bound (upper *and* lower) on complexity of an algorithm
- Iff an algorithm is  $O(f(n))$  and also  $\Omega(f(n))$ , then it is  $\Theta(f(n))$
- $\Theta(f(n))$  means asymptotically not less or more than  $f(n)$  steps
- *Examples:*
  - An algorithm that is  $O(n)$  is also  $O(n^2)$
  - One that is  $\Theta(n)$  is too fast to be  $\Theta(n^2)$
  - One that is  $\Theta(n^2)$  is too slow to be  $\Theta(n)$

## Formal definitions of $O$ , $\Omega$ , $\theta$

- These define *sets of functions that increase at a similar rate*
- $f \in O(g)$  iff  
 $(\exists c, k) (\forall x) f(x) \leq cg(x) + k$
- $f(x) \in \Omega(g(x))$  iff  
 $\exists c, n_0$  s.t.  $f(x) \geq cg(x) + n_0$
- $f(x) \in \theta(g(x))$  iff  
 $f(x) \in O(g(x)) \wedge f(x) \in \Omega(g(x))$

### Example

$f \in O(g)$  because there exist particular constants  $c, k$ , s.t. for all  $n > k, f(n) \leq cg(n)$

David Keil      Analysis of Algorithms      3. Recurrences      1/11      13

## Notation conventions

*Functions:*

- We often write  $f(n)$  for  $f: \mathbb{N} \rightarrow \mathbb{N}$
- $f(n) \in \theta(n^d)$  means “ $f: \mathbb{N} \rightarrow \mathbb{N}$  is in efficiency class  $\theta(g(n))$ , where  $g(n) = n^d$ , i.e.,  $g$  is a  $d$ -degree polynomial”
- *Example:* if  $f(n) = 5n^2 - 80$ , then  $f(n) \in \theta(n^2)$

*Arrays:*

- $A[1..|A|-1]$  means the sequence in  $A$  from the first to the next-to-last
- $A + \langle x \rangle$  means array  $A$  concatenated with the array consisting of the element value  $x$

David Keil      Analysis of Algorithms      3. Recurrences      1/11      14

## 2. Algorithm analysis

- *Analysis*: separation into components
- Form of running-time analysis for algorithm  $\alpha$  with running time that is big-O or big- $\Theta$  (theta) of  $f$ , where  $f$  is a function of the size of  $\alpha$ 's input:

$$T_{\alpha}(n) = O(f(n))$$

- *Examples*:
  - $T_{\text{Array-append}}(n) = \Theta(1)$
  - $T_{\text{Linear-search}}(n) = \Theta(n)$
  - ... because appending a new element to an array takes constant time and linear search takes time proportional to the array size

## Assumptions about hardware

*We assume that*

- steps involving single machine instructions, e.g., assignment, arithmetic operations, comparison, take one time unit each
- architecture is *random-access machine*, i.e., variable access is constant-time, including for array elements
- I/O steps take constant time

## Best, worst, and average cases

- *Best* or *worst* case specifies efficiency with input data that yields lowest/highest running time
- *Example:*
  - Best case for linear search is when  $key = A[1]$
  - Worst case is key in last position
- *Average* case (expected running time) is not necessarily the average of best and worst cases

## Empirical analysis of algorithms

- *Time* algorithm or *count steps*, for different quantities of input
- Plot time vs. data quantity as scatterplot
- Interpolate/extrapolate to derive time function
- *Exercise:* Plot running time of Quicksort for data sizes of 100, 200, 400, 800, 1600, 3200
- Our emphasis is *theoretical* analysis (whose usefulness is tested empirically)

## Bubble sort

```
Repeat
  swap ← false
  for i ← 1 to size(A) – 1
    if A[ i ] > A[ i + 1 ]
      swap (A[ i ], A[i +1])
      swap ← true
until swap = false
```

- The use of nested loops suggests what about running time, for an  $n$ -element array?
- Challenge: Write a recursive version of *Bubble*

## Loops and complexity

- A single loop of  $n$  iterations, where each iteration executes  $O(1)$  steps, is  $O(n)$
- A loop nested to two levels, each with roughly  $n$  iterations, where each iteration executes  $O(1)$  steps, is  $O(n^2)$
- If we start with  $n$  items to look at and cut our remaining work in half at each step, then the job will take  $O(\log_2 n)$  such steps.
- If our loops are nested to  $n$  levels, as in password guessing, then algorithm is  $O(2^n)$  — offer job to someone else

## Slower part of an algorithm dominates its running time

- *Theorem:*  
 $(\forall n) T_1(n) \in O(g_1(n)) \wedge T_2(n) \in O(g_2(n)) \Rightarrow T_1(n) + T_2(n) \in O(\max\{g_1(n), g_2(n)\})$
- *Discussion:* The times of two steps of an algorithm, added together, grow as the order of the *slower* (maximum-time) part
- *Example:* Algorithm that checks for duplicates in an array by bubble-sorting then checking consecutive elements, is  $O(n^2)$

## Meaning of $O$ , $\Theta$

- $O()$ ,  $\Theta()$  expressions denote *sets of functions*
- In algorithm analysis, this is used with *time functions for algorithms*
- So, for example,  $\Theta(n)$  is not “slower” than  $\Theta(\lg n)$ , even though time functions with respective definitions characterize slower algorithms.

## Comparing orders of growth

- Let  $q = \lim_{n \rightarrow \infty} (T(n) / g(n))$
- If  $q = 0$  then  $T(n) \in O(g(n))$
- If  $q$  is constant  $> 0$  then  $T(n) \in \theta(g(n))$
- If  $q = \infty$  then  $T(n) \in \Omega(g(n))$
- *Example:* let  $T(n) = \log_2 n$ ,  $g(n) = \text{sqrt}(n)$ ;  
 $\lim_{n \rightarrow \infty} \log_2 n = 2 \log_2 e \lim_{n \rightarrow \infty} \text{sqrt}(n) / n$   
 $= 0$ ;  
hence  $\log_2 n \in O(\text{sqrt}(n))$

## Decision trees and running time

- Execution of an algorithm may be represented as descending a *decision tree*, from root to leaf
- *Examples:*
  - Binary search [Johnsonbaugh, p. 168]
  - Sort [p. 255]
- *Theorems:*
  - Worst-case time for comparison-based search of sorted array is  $\Omega(\lg n)$ , hence binary search is asymptotically optimal
  - Worst-case time for comparison-based sorting is  $\Omega(n \lg n)$

## 3. Recurrence relations and time functions

- Recurrences express functions, algorithms, and the running-time functions for the algorithms
- A *recurrence* may be used to define a function inductively (recursively) by giving the base-case definition and the recursive-case definition
- Likewise a *time recurrence* defines the time function that corresponds to a recursive algorithm
- A formula defines how to go from the recurrence that expresses the algorithm to a time recurrence

## Peano's axioms: definition by induction

1. 0 is a natural number ( $0 \in \mathbf{N}$ )
2. Every natural number  $n$  has a unique successor,  $n'$ , also a natural number  
( $\forall n \in \mathbf{N}$ )  $n' \in \mathbf{N}$
3. All natural numbers follow (1) or (2)  
( $\forall n \in \mathbf{N}$ )  $n = 0 \vee \exists m \in \mathbf{N}$  s.t.  $n = m'$

- *Significance:* These axioms, or assumptions, provide a formal logical basis to work with counting numbers. *Note:* (2) is recursive
- Computation is a formal way to manipulate numbers and objects representable by them.

### Recurrence relations

- *Definition:* A set of tuples that define a sequence by relating an element to some of its predecessors
- *Example:* Fibonacci sequence, where  $f_1 = f_2 = 1, f_n = f_{n-1} + f_{n-2}$  for  $n \geq 3$
- *Recurrence relations* define the computable functions  $f: \mathbf{N} \rightarrow \mathbf{N}$ ,  $\phi: \Sigma^* \rightarrow \Sigma^*$
- *Notation:* as above or as on next slide

### A recurrence may define a function algorithmically

$$\text{sum}(a, b) = \begin{cases} a & \text{if } b = 0 \\ \text{sum}(\text{succ}(a), \text{pred}(b)) & \text{otherwise} \end{cases}$$

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

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n \times \text{factorial}(n-1) & \text{otherwise} \end{cases}$$

## Defining *max* for arrays

(a) Set definition:  $\max(A) = x \in A$  s.t.  $(\forall i \leq |A|) x \geq A[i]$

(b) Algorithmic definition for array:

**Max-elt**(A) =

$$\begin{cases} \uparrow & \text{if } |A| = 0 \\ A[1] & \text{if } |A| = 1 \\ A[1] & \text{if } A[1] > \text{Max-elt}(A[2 .. |A|]) \\ \text{Max-elt}(A[2 .. |A|]) & \text{otherwise} \end{cases}$$

$$T_{\text{Max-elt}}(n) = \begin{cases} O(1) & \text{if } n = 0, 1 \\ O(1) + 2T_{\text{Max-elt}}(n - 1) & \text{otherwise} \end{cases}$$

## Converting associative binary operators to *n*-ary operators

- If  $\oplus$  is an associative binary operator on values in set  $S$  (stored as an array), then  $f: S^n \rightarrow S$  is the corresponding operator on arrays of  $S$ , where

$$f(A) = \begin{cases} \uparrow & \text{if } |A| = 0 \\ A[1] & \text{if } |A| = 1 \\ A[1] \oplus f(A[2 .. |A|]) & \text{otherwise} \end{cases}$$

## Recurrences and time analysis

- Recurrences are function definitions
- Their form is different from pseudocode
  - There are curly braces in recurrences
  - In recurrences, the return value goes on the left and the IF goes on the right.
- In running-time recurrences, the solution is a big-O expression.

## Russian peasants' algorithm

### Product (a, b)

```
result ← 0
while a > 0
  if a is odd
    add b to result
  a ← ⌊a ÷ 2⌋
  b ← Product(b, 2)
Return result
```

Example:  $13 \times 5 = 65$

	<i>a</i>	<i>b</i>	<i>product</i>
			0
1.	13	5	5
2.	6	10	5
3.	3	20	25
4.	1	40	65
5.	0	80	

## Algorithm analysis using recurrences

1. Define algorithm iteratively in pseudocode
2. Define recurrence relation that specifies function computed by algorithm
3. Based on (2), define recurrence that relates running time of algorithm to size of input
4. Solve recurrence relation in  $O$ ,  $\Omega$ ,  $\theta$  notation, using the notion of depth of recursion (number of times algorithm invokes self)

## Example problem: array search

1. Iterative version:

Search(A, key)

```
i ← 1
While i ≤ |A|
  if A[ i ] = key return true
  i ← i + 1
Return false
```

2. Recursive definition:

*Search*(*A*, *key*) =

{	False	if   <i>A</i>   = 0
	True	if <i>A</i> [1] = <i>key</i>
	<i>Search</i> ( <i>A</i> [2 ..   <i>A</i>  ], <i>key</i> )	otherwise

#### 3. Time recurrence

- It takes one step to test size of array,
- plus one to compare  $A[1]$  to  $key$ ,
- plus the number of steps to search the rest of the array, starting at 2

$$T_{Search}(n) = \begin{cases} 2 & \text{if } n \leq 1 \\ 2 + T_{Search}(n-1) & \text{otherwise} \end{cases}$$

#### 4. Solution to time recurrence: $O(n)$

because time for base case is constant ( $O(1)$ ) and depth of recursion is  $(n-1)$ , i.e.,  $O(n)$  and  $O(n) \times O(1) = O(n)$

## 4. Solving recurrences

- Recurrences express computable function definitions, algorithms, and the running-time functions of the algorithms
- Formula for analysis of recursive algorithms:

$$T_{Algorithm-A}(n) = O(\text{Base-case-runtime} \times \text{Depth-of-recursion})$$

- *Example:*

$$T_{Quicksort}(n) = O(n) \times O(\lg n) = O(n \lg n)$$

for average case, because partition (base case) is  $O(n)$  and depth of recursion is logarithmic

## Linear time

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + T(n - 1) & \text{otherwise} \end{cases}$$

**Solution:**  $O(n)$ , because one step is needed for base case at each level of recursion, and there are  $(n - 1)$  levels

*Examples:* Evaluation of each function on previous slide takes one step for base case and depth of recursion is  $O(n)$

## Recursive list traversal

### Traverse (Node-ptr)

If *Node-ptr* not null

Visit node pointed to by *Node-ptr*

Traverse (*Next (Node-ptr)*)

- *Complexity:*  $T(n) = \begin{cases} 1 & \text{for null list} \\ 2 + T(n - 1) & \text{otherwise} \end{cases}$
- *Solution:*  $O(n)$  is running time of *Traverse*, assuming constant time of visiting one node and assuming constant time of *next* method

## Brute-force algorithms

1. Numeric computations
2. Vector traversal
3. Sorting
4. 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 in general case
- *Topic objective:* Explain the brute force design approach to algorithmic problems, and code solutions designed under it

## Objectives

- 3c. Write a recurrence that defines the function computed by a brute-force algorithm
- 3d. Write and solve a time recurrence for a brute-force algorithm
- 3e. Explain and use the brute force approach, and analyze solutions designed under it

## 1. Numeric computations

### Multiplication and exponentiation

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

## Decrease by one

Decrease-by-one algorithms have recurrences that reduce the quantity in the recursive case by one at each step

### Factorial (n)

```
if  $n \leq 1$ 
    return 1
else
    return  $n \times \text{Factorial}(n - 1)$ 
```

### Fibonacci (n)

```
if  $n < 2$ 
    return 1
else
    return  $\text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$ 
```

**Analysis:**  
*Factorial is  $O(n)$*   
*Fibonacci is exponential-time*

## Solving recurrences: decrease-by-one

- Decrease-by-one algorithms exploit the 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
- $T(n) = T(n-1) + f(n)$   
 $= T(n-2) + f(n-1) + f(n) \dots$

## 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

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

Running time:

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

### Adding elements of an array

#### Array-sum(A)

$i \leftarrow 1$

$y \leftarrow 0$

while  $i \leq |A|$

    // LI:  $y = A[1] + A[2] + \dots + A[i-1]$

$y \leftarrow y + A[i]$

$i \leftarrow i + 1$

return  $y$

    // Post:  $y = A[1] + A[2] + \dots + A[|A|]$

## Analysis of array-sum

Recurrence for *Array-sum*:

$$\begin{aligned} \text{Sum}(A) = & \\ \{ 0 & \text{ if } |A| = 0 \\ \{ A[1] & \text{ if } |A| = 1 \\ \{ A[1] + \text{Sum}(A[2 .. |A|]) & \text{ otherwise} \end{aligned}$$

Time recurrence:

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

## Brute-force string search

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

> Returns location of first occurrence of *key* in *S*

for  $i \leftarrow 1$  to  $n - m + 1$  do

$j \leftarrow 1$

    while  $j \leq m \wedge \text{key}[j] = S[i + j]$

$j \leftarrow j + 1$

    if  $j > m$  return  $i$

return  $-1$

This algorithm checks at each location in *S* to tell whether the substring at that location matches *key*

## Recurrences for string search

Algorithms with nested loops require two or more recurrences.

*Match* does character by character comparison:

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

*Search* tells whether S2 can be found in S1:

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

## Longest common subsequence

- *Problem:* Given sequences  $x_1, x_2$ , what is the longest subsequence  $y$  s.t.  $y$  is a subsequence of both  $x_1$  and  $x_2$ ?
- Elements of subsequences are not necessarily contiguous, e.g., “dab” is a subsequence of “database”
- *Brute-force solution:* For each subsequence of  $x_1$ , search for it in  $x_2$ , storing the longest found as this process continues and replacing it as a longer common subsequence is found
- *Time:* exponential because of the number of subsequences

## 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*

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

## 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) =**

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

**Tinsert-asc(n) =**

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

David Keil      Analysis of Algorithms      3. Recurrences      1/11      55

## Merging two sorted arrays

Repeat until A or B is exhausted:

- $x \leftarrow \min \{A[ai], B[bi]\}$
- append x to the end of C, incrementing ci
- increment ai or bi as appropriate

Copy remainder of A or B to end of C

- C should be as large as A, B, together

David Keil      Analysis of Algorithms      3. Recurrences      1/11      56

### 3. Sorting

**Insertion-sort (A)**

```
i ← 1  
while i < |A|  
  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

David Keil Analysis of Algorithms 3. Recurrences 1/11 57

### Recursive insertion sort

**Insertion-sort (A)**

```
If |A| ≤ 1  
  return A  
Else  
  return Insert-ascending  
    (Insertion-sort(A [1 .. |A| - 1]), A[|A|])
```

This algorithm recursively sorts all but the last element the array and then inserts the last element into the sorted result

David Keil Analysis of Algorithms 3. Recurrences 1/11 58

## Recurrence for insertion sort

### Insertion-sort (A) =

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

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

## Selection Sort, recursive version

### Selection-sort (A, start, size)

If size > 1

Find lowest value in A[start..size]

Swap it with A[start]

Return Selection-sort (A, start + 1, size)

else return A

*Time-complexity recurrence:*

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

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

### Bubble-sort(A)

**Precondition:** A is an array

$num\_passes \leftarrow 0$

Repeat

**Invariant:** Rightmost ( $num\_passes$ ) elements are in ascending order

$swapped \leftarrow false$

for  $i \leftarrow 1$  to  $size(A) - 1$

**Invariant:**  $A[i]$  is not smaller than any in  $A[0..i-1]$

if  $A[i] > A[i + 1]$

$swap(A[i], A[i + 1])$

$swapped \leftarrow true$

$num\_passes \leftarrow num\_passes + 1$

until  $swapped$  is *false*

**Postcondition:** A is in ascending order

### Bubble sort

$Max\text{-at-right}(A) =$

$$\begin{cases} A & \text{if } |A| \leq 1 \\ A[1] \text{ Max-at-right}(A[2 .. |A|]) & \text{if } A[1] < A[2] \\ A[2] \text{ Max-at-right}(A[1]A[3 .. |A|]) & \text{otherwise} \end{cases}$$

- Returns A, modified so that max element is at right

$Bubble(A) =$

$$\begin{cases} A & \text{if } |A| \leq 1 \\ Bubble(Max\text{-at-right}(A) [1 .. |A|-1]) \\ \quad +Max(A) & \text{otherwise} \end{cases}$$

- Returns *Bubble-sort* of A, minus the maximum, followed by max element

## Solving recurrences for quadratic-time algorithms

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \theta(n) + T(n - 1) & \text{otherwise} \end{cases}$$

**Solution:**  $\theta(n^2)$ , because  $n$  steps are needed for each of  $(n-1)$  levels of recursion

*Examples:* bubble sort, selection sort, insertion sort

## 4. Exhaustive search

- Generate each possible solution to problem; select one that satisfies constraint
- *Optimization version:* 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
- *Analysis:* running time is exponential in number of cities

## Exponential-time algorithms

- Exhaustive search often requires time *exponential* in the size of the problem
- *Examples:* Towers of Hanoi/Brahma, TSP, SAT, knapsack problem
- *Double recursion* (e.g., Fibonacci) may take exponential time
- These algorithms may be necessary to solve problems in which *combinatorial explosion* occurs

## 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

### Closest pair ( $P [ i, j ]$ )

```
If n < 2 throw exception
max-dist ← Distance(1, 2)
closest-pair ← (1, 2)
for i ← 1 to n - 1
  for j ← i + 1 to n
    if dist(P, i, j) < max-dist
      max-dist ← Distance(P, i, j)
      closest-pair ← (P, i, j)
Return closest-pair
```

Time:  $O(n^2)$

```
Distance (A, i, j)
Return sqrt(
  (A[ i ].x - A[ j ].x)2 +
  (A[ i ].y - A[ j ].y)2
)
```

### 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

## Knapsack Problem

- *Problem:* find maximum-valued subset of weighted items under a given total weight.
- *Brute-force solution:* Find total weight of each subset, find subset with maximum value where weight does not exceed maximum weight
- *Analysis:* time exponential in number of elements of set, since the power set of an  $n$ -set has  $2^n$  elements

## Hanoi (*source, dest, intermed, ndisks*)

```

If ndisks > 0
  Hanoi (source, intermed, dest, ndisks-1)
  Move top disk from source to dest
  Hanoi (intermed, dest, source, ndisks-1)
    
```

(Move the top  $(n - 1)$  disks to the middle peg, then move bottom disk to the goal peg, then move all  $(n - 1)$  disks on the middle peg to the goal.)

*Recurrence, expressing number of steps:*

$$T_{Hanoi}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 T_{Hanoi}(n - 1) + 1 & \text{if } n > 1 \end{cases}$$

Make 2 recursive calls
Move 1 disk

### Complexity of Hanoi solution

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

- Recurrence suggests that time doubles on each recursive step, so guess running time is  $O(2^n - 1)$
- Proof by induction:
  - It works for  $n = 1$ :  $2^n - 1 = 2 - 1 = 1$
  - If  $T(n - 1) = 2^n - 1 - 1$  then  $T(n) = 2^n - 1$ :
 
$$\begin{aligned} T(n) &= 2T(n - 1) + 1 && \text{Per recurrence} \\ &= 2(2^n - 1 - 1) + 1 && \text{Guess} \\ &= 2^n - 2 + 1 && \text{Add exponents} \\ &= 2^n - 1 && \text{(QED)} \end{aligned}$$

### Solving double recursion

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + 2T(n - 1) & \text{otherwise} \end{cases}$$

**Solution:**  $O(2^n)$ , because at each of  $(n-1)$  recursive steps the amount of time required doubles

*Example:* Towers of Hanoi/Brahma

## Problems with formulas in logic

Given a propositional logic formula,

- Does it hold, given a set of variable assignments?  
*O(n) brute-force solution:* evaluate the formula under the given interpretation
- Is it a *tautology* (valid under all variable assignments)?
- Is it a *contradiction* (valid under no assignments)?
- Is it *satisfiable*?  
*Brute-force solution:* Generate a truth table, with  $2^n$  rows for  $n$  variables, see if any rows evaluate to *true*

## Satisfiability brute-force solution

### Satisfiable ( $\phi, t$ )

> Pre:  $\phi$  is a formula in propositional logic,

> with truth assignments  $t_1, t_2, \dots, t_n$ ,

$y \leftarrow \text{false}$

for  $i \leftarrow 1$  to  $2^{|\phi|}$

> LI:  $y$  is true iff  $\phi$  with var asgt indicated by

> some bit pattern in  $(0 \dots i - 1)$  evaluates to true

if  $\text{eval}(\phi, i) = \text{true}$

$y \leftarrow \text{true}$

return  $y$

> Post:  $y$  is true iff  $\phi$  with var asgt indicated by

> some bit pattern in  $(0 \dots 2^{|\phi|})$  evaluates to true

## Optimization problems

- Many problems require finding the *best* solution in a very large ( $O(2^n)$ ) set of possible solutions
- Brute force (exhaustive search) offers an easily designed but long-running-time approach
- Divide and conquer, greedy, dynamic programming, and approximation approaches are often more efficient

## Terminology

- |                                  |                             |
|----------------------------------|-----------------------------|
| algorithm analysis               | linear time $O(n)$          |
| asymptotic behavior              | logarithmic time $O(\lg n)$ |
| average case                     | lower bound                 |
| best case                        | Master Theorem              |
| big-O notation                   | quadratic time: $O(n^2)$    |
| big-Omega notation               | recurrence relation         |
| constant time $O(1)$             | running time                |
| decision tree                    | running-time recurrence     |
| depth of recursion               | theta notation              |
| double recursion                 | tight bound                 |
| empirical analysis of algorithms | time function               |
| expected running time            | upper bound                 |
| exponential time: $O(2^n)$       | worst case                  |

**Concepts**

brute force	<i>Drag-max</i>
brute-force closest-pair	exhaustive search
brute-force convex hull	<i>Insert-ascending</i>
brute-force knapsack	insertion sort
brute-force Traveling Salesperson	linear search
Bubble sort	matrix multiplication
	Towers of Hanoi (Brahma)

David Keil Analysis of Algorithms 3. Recurrences 1/11 77

**References**

Cormen, Leiserson, Rivest. *Introduction to Algorithms*. MIT Press, 1990.

Hamburger and Richards.

A. Levitin. *The Design and Analysis of Algorithms*. Addison Wesley, 2003. Chapters 2, 4-5

R. Johnsonbaugh and M. Schaefer. *Algorithms*. Pearson Prentice Hall, 2004. Chapter 2.

David Keil Analysis of Algorithms 3. Recurrences 1/11 78

## 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.