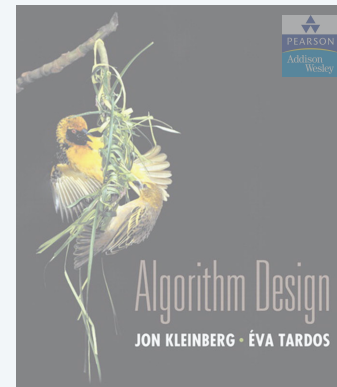


## 2. ALGORITHM ANALYSIS

- computational tractability
- asymptotic order of growth
- implementing Gale–Shapley
- survey of common running times

Lecture slides by Kevin Wayne  
 Copyright © 2005 Pearson–Addison Wesley  
<http://www.cs.princeton.edu/~wayne/kleinberg-tardos>

Last updated on 5/5/18 5:36 AM



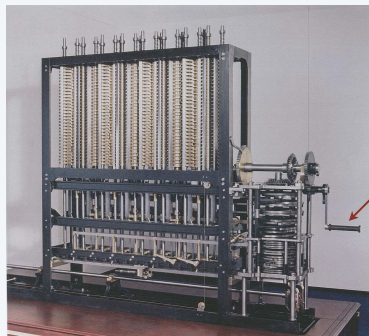
## 2. ALGORITHM ANALYSIS

- computational tractability
- asymptotic order of growth
- implementing Gale–Shapley
- survey of common running times

SECTION 2.1

## A strikingly modern thought

“As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?” — Charles Babbage (1864)

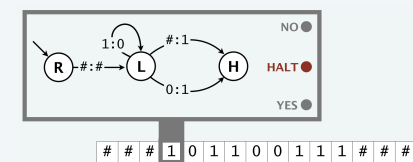


Analytic Engine

how many times do you have to turn the crank?

## Models of computation: Turing machines

**Deterministic Turing machine.** Simple and idealistic model.



**Running time.** Number of steps.

**Memory.** Number of tape cells utilized.

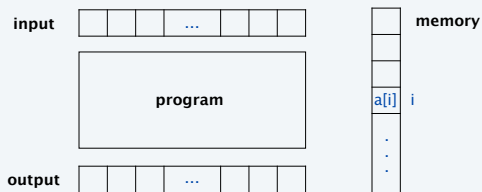
**Caveat.** No random access of memory.

- Single-tape TM requires  $\geq n^2$  steps to detect  $n$ -bit palindromes.
- Easy to detect palindromes in  $\leq cn$  steps on a real computer.

## Models of computation: word RAM

### Word RAM.

- Each memory location and input/output cell stores a  $w$ -bit integer.
- Primitive operations: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, ...



**Running time.** Number of primitive operations.

**Memory.** Number of memory cells utilized.

**Caveat.** At times, need more refined model (e.g., multiplying  $n$ -bit integers).

5

## Brute force

**Brute force.** For many nontrivial problems, there is a natural brute-force search algorithm that checks every possible solution.

- Typically takes  $2^n$  steps (or worse) for inputs of size  $n$ .
- Unacceptable in practice.



**Ex.** Stable matching problem: test all  $n!$  perfect matchings for stability.

6

## Polynomial running time

**Desirable scaling property.** When the input size doubles, the algorithm should slow down by at most some constant factor  $C$ .

**Def.** An algorithm is **poly-time** if the above scaling property holds.

There exist constants  $c > 0$  and  $d > 0$  such that, for every input of size  $n$ , the running time of the algorithm is bounded above by  $c n^d$  primitive computational steps.

choose  $C = 2^d$



von Neumann  
(1953)



Nash  
(1955)



Gödel  
(1956)



Cobham  
(1964)



Edmonds  
(1965)



Rabin  
(1966)

7

## Polynomial running time

We say that an algorithm is **efficient** if it has a polynomial running time.

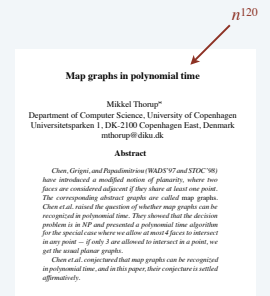
**Theory.** Definition is (relatively) insensitive to model of computation.

**Practice.** It really works!

- The poly-time algorithms that people develop have both small constants and small exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

**Exceptions.** Some poly-time algorithms in the wild have galactic constants and/or huge exponents.

**Q.** Which would you prefer:  $20n^{120}$  or  $n^{1+0.02 \ln n}$ ?



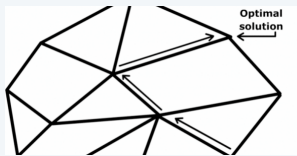
8

## Worst-case analysis

**Worst case.** Running time guarantee for **any input** of size  $n$ .

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

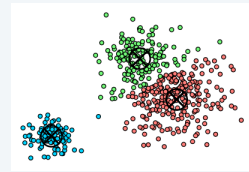
**Exceptions.** Some exponential-time algorithms are used widely in practice because the worst-case instances don't arise.



simplex algorithm



Linux grep



k-means algorithm

9

## Other types of analyses

**Probabilistic.** Expected running time of a **randomized algorithm**.

**Ex.** The expected number of compares to quicksort  $n$  elements is  $\sim 2n \ln n$ .



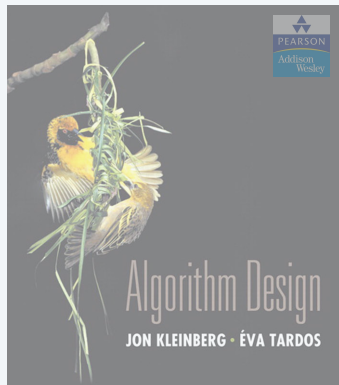
**Amortized.** Worst-case running time for **any sequence** of  $n$  operations.

**Ex.** Starting from an empty stack, any sequence of  $n$  push and pop operations takes  $O(n)$  primitive computational steps using a resizing array.



**Also.** Average-case analysis, smoothed analysis, competitive analysis, ...

10



SECTION 2.2

## 2. ALGORITHM ANALYSIS

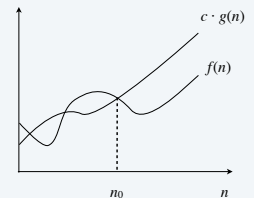
- ▶ *computational tractability*
- ▶ *asymptotic order of growth*
- ▶ *implementing Gale-Shapley*
- ▶ *survey of common running times*

## Big O notation

**Upper bounds.**  $f(n)$  is  $O(g(n))$  if there exist constants  $c > 0$  and  $n_0 \geq 0$  such that  $0 \leq f(n) \leq c \cdot g(n)$  for all  $n \geq n_0$ .

**Ex.**  $f(n) = 32n^2 + 17n + 1$ .

- $f(n)$  is  $O(n^2)$ . ← choose  $c = 50, n_0 = 1$
- $f(n)$  is neither  $O(n)$  nor  $O(n \log n)$ .



**Typical usage.** Insertion sort makes  $O(n^2)$  compares to sort  $n$  elements.

12

## Analysis of algorithms: quiz 1



Let  $f(n) = 3n^2 + 17n \log_2 n + 1000$ . Which of the following are true?

- A.  $f(n)$  is  $O(n^2)$ .
- B.  $f(n)$  is  $O(n^3)$ .
- C. Both A and B.
- D. Neither A nor B.

13

## Big O notational abuses

**One-way “equality.”**  $O(g(n))$  is a set of functions, but computer scientists often write  $f(n) = O(g(n))$  instead of  $f(n) \in O(g(n))$ .

**Ex.** Consider  $g_1(n) = 5n^3$  and  $g_2(n) = 3n^2$ .

- We have  $g_1(n) = O(n^3)$  and  $g_2(n) = O(n^3)$ .
- But, do not conclude  $g_1(n) = g_2(n)$ .

**Domain and codomain.**  $f$  and  $g$  are real-valued functions.

- The domain is typically the natural numbers:  $\mathbb{N} \rightarrow \mathbb{R}$ .
- Sometimes we extend to the reals:  $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$ . ↖ input size, recurrence relations
- Or restrict to a subset. ↖ plotting, limits, calculus

**Bottom line.** OK to abuse notation in this way; not OK to misuse it.

14

## Big O notation: properties

**Reflexivity.**  $f$  is  $O(f)$ .

**Constants.** If  $f$  is  $O(g)$  and  $c > 0$ , then  $cf$  is  $O(g)$ .

**Products.** If  $f_1$  is  $O(g_1)$  and  $f_2$  is  $O(g_2)$ , then  $f_1 f_2$  is  $O(g_1 g_2)$ .

**Pf.**

- $\exists c_1 > 0$  and  $n_1 \geq 0$  such that  $0 \leq f_1(n) \leq c_1 \cdot g_1(n)$  for all  $n \geq n_1$ .
- $\exists c_2 > 0$  and  $n_2 \geq 0$  such that  $0 \leq f_2(n) \leq c_2 \cdot g_2(n)$  for all  $n \geq n_2$ .
- Then,  $0 \leq f_1(n) \cdot f_2(n) \leq \frac{c_1 \cdot c_2}{c} \cdot g_1(n) \cdot g_2(n)$  for all  $n \geq \max\{n_1, n_2\}$ . ■

**Sums.** If  $f_1$  is  $O(g_1)$  and  $f_2$  is  $O(g_2)$ , then  $f_1 + f_2$  is  $O(\max\{g_1, g_2\})$ .

↖ ignore lower-order terms

**Transitivity.** If  $f$  is  $O(g)$  and  $g$  is  $O(h)$ , then  $f$  is  $O(h)$ .

**Ex.**  $f(n) = 5n^3 + 3n^2 + n + 1234$  is  $O(n^3)$ .

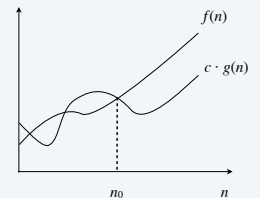
15

## Big Omega notation

**Lower bounds.**  $f(n)$  is  $\Omega(g(n))$  if there exist constants  $c > 0$  and  $n_0 \geq 0$  such that  $f(n) \geq c \cdot g(n) \geq 0$  for all  $n \geq n_0$ .

**Ex.**  $f(n) = 32n^2 + 17n + 1$ .

- $f(n)$  is both  $\Omega(n^2)$  and  $\Omega(n)$ . ↖ choose  $c = 32, n_0 = 1$
- $f(n)$  is not  $\Omega(n^3)$ .



**Typical usage.** Any compare-based sorting algorithm requires  $\Omega(n \log n)$  compares in the worst case.

**Vacuous statement.** Any compare-based sorting algorithm requires at least  $O(n \log n)$  compares in the worst case.

16

## Analysis of algorithms: quiz 2



Which is an equivalent definition of big Omega notation?

- A.  $f(n)$  is  $\Omega(g(n))$  iff  $g(n)$  is  $O(f(n))$ .
- B.  $f(n)$  is  $\Omega(g(n))$  iff there exist constants  $c > 0$  such that  $f(n) \geq c \cdot g(n) \geq 0$  for infinitely many  $n$ .
- C. Both A and B.
- D. Neither A nor B.

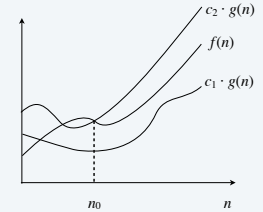
17

## Big Theta notation

**Tight bounds.**  $f(n)$  is  $\Theta(g(n))$  if there exist constants  $c_1 > 0$ ,  $c_2 > 0$ , and  $n_0 \geq 0$  such that  $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  for all  $n \geq n_0$ .

**Ex.**  $f(n) = 32n^2 + 17n + 1$ .

- $f(n)$  is  $\Theta(n^2)$ . ← choose  $c_1 = 32$ ,  $c_2 = 50$ ,  $n_0 = 1$
- $f(n)$  is neither  $\Theta(n)$  nor  $\Theta(n^3)$ .



**Typical usage.** Mergesort makes  $\Theta(n \log n)$  compares to sort  $n$  elements.

↗  
between  $\frac{1}{2} n \log_2 n$   
and  $n \log_2 n$

19

## Analysis of algorithms: quiz 3



Which is an equivalent definition of big Theta notation?

- A.  $f(n)$  is  $\Theta(g(n))$  iff  $f(n)$  is both  $O(g(n))$  and  $\Omega(g(n))$ .
- B.  $f(n)$  is  $\Theta(g(n))$  iff  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for some constant  $0 < c < \infty$ .
- C. Both A and B.
- D. Neither A nor B.

20

## Asymptotic bounds and limits

**Proposition.** If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for some constant  $0 < c < \infty$  then  $f(n)$  is  $\Theta(g(n))$ .

**Pf.**

- By definition of the limit, for any  $\epsilon > 0$ , there exists  $n_0$  such that

$$c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon$$

for all  $n \geq n_0$ .

- Choose  $\epsilon = \frac{1}{2} c > 0$ .
- Multiplying by  $g(n)$  yields  $\frac{1}{2} c \cdot g(n) \leq f(n) \leq \frac{3}{2} c \cdot g(n)$  for all  $n \geq n_0$ .
- Thus,  $f(n)$  is  $\Theta(g(n))$  by definition, with  $c_1 = \frac{1}{2} c$  and  $c_2 = \frac{3}{2} c$ . ■

**Proposition.** If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , then  $f(n)$  is  $O(g(n))$  but not  $\Omega(g(n))$ .

**Proposition.** If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ , then  $f(n)$  is  $\Omega(g(n))$  but not  $O(g(n))$ .

21

## Asymptotic bounds for some common functions

**Polynomials.** Let  $f(n) = a_0 + a_1 n + \dots + a_d n^d$  with  $a_d > 0$ . Then,  $f(n)$  is  $\Theta(n^d)$ .

**Pf.**

$$\lim_{n \rightarrow \infty} \frac{a_0 + a_1 n + \dots + a_d n^d}{n^d} = a_d > 0$$

**Logarithms.**  $\log_a n$  is  $\Theta(\log_b n)$  for every  $a > 1$  and every  $b > 1$ .

**Pf.**

$$\frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}$$

no need to specify base  
(assuming it is a constant)

**Logarithms and polynomials.**  $\log_a n$  is  $O(n^d)$  for every  $a > 1$  and every  $d > 0$ .

**Pf.**

$$\lim_{n \rightarrow \infty} \frac{\log_a n}{n^d} = 0$$

**Exponentials and polynomials.**  $n^d$  is  $O(r^n)$  for every  $r > 1$  and every  $d > 0$ .

**Pf.**

$$\lim_{n \rightarrow \infty} \frac{n^d}{r^n} = 0$$

**Factorials.**  $n!$  is  $2^{\Theta(n \log n)}$ .

**Pf.** Stirling's formula:  $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

22

## Big O notation with multiple variables

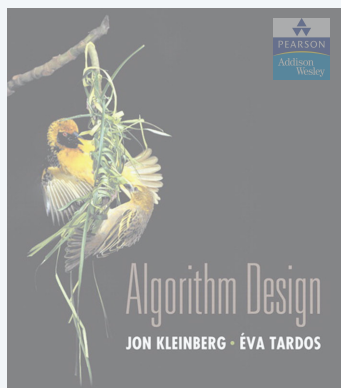
**Upper bounds.**  $f(m, n)$  is  $O(g(m, n))$  if there exist constants  $c > 0$ ,  $m_0 \geq 0$ , and  $n_0 \geq 0$  such that  $f(m, n) \leq c \cdot g(m, n)$  for all  $n \geq n_0$  and  $m \geq m_0$ .

**Ex.**  $f(m, n) = 32mn^2 + 17mn + 32n^3$ .

- $f(m, n)$  is both  $O(mn^2 + n^3)$  and  $O(mn^3)$ .
- $f(m, n)$  is neither  $O(n^3)$  nor  $O(mn^2)$ .

**Typical usage.** Breadth-first search takes  $O(m + n)$  time to find a shortest path from  $s$  to  $t$  in a digraph with  $n$  nodes and  $m$  edges.

23



SECTION 2.3

## 2. ALGORITHM ANALYSIS

- ▶ computational tractability
- ▶ asymptotic order of growth
- ▶ implementing Gale-Shapley
- ▶ survey of common running times

## Efficient implementation

**Goal.** Implement Gale-Shapley to run in  $O(n^2)$  time.

**GALE-SHAPLEY** (preference lists for  $n$  hospitals and  $n$  students)

**INITIALIZE**  $M$  to empty matching.

**WHILE** (some hospital  $h$  is unmatched)

$s \leftarrow$  first student on  $h$ 's list to whom  $h$  has not yet proposed.

**IF** ( $s$  is unmatched)

Add  $h$ - $s$  to matching  $M$ .

**ELSE IF** ( $s$  prefers  $h$  to current partner  $h'$ )

Replace  $h'$ - $s$  with  $h$ - $s$  in matching  $M$ .

**ELSE**

$s$  rejects  $h$ .

**RETURN** stable matching  $M$ .

25

## Efficient implementation

**Goal.** Implement Gale–Shapley to run in  $O(n^2)$  time.

**Representing hospitals and students.** Index hospitals and students  $1, \dots, n$ .

**Representing the matching.**

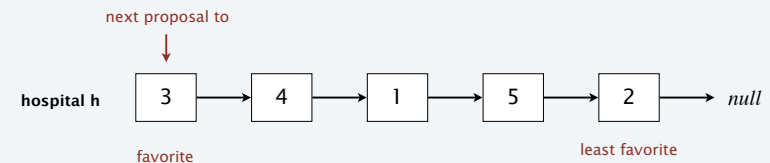
- Maintain two arrays  $student[h]$  and  $hospital[s]$ .
  - if  $h$  matched to  $s$ , then  $student[h] = s$  and  $hospital[s] = h$
  - use value 0 to designate that hospital or student is unmatched
- Can add/remove a pair from matching in  $O(1)$  time.
- Maintain set of unmatched hospitals in a queue (or stack).
- Can find an unmatched hospital in  $O(1)$  time.

26

## Data representation: making a proposal

**Hospital makes a proposal.**

- Key operation: find hospital's next favorite student.
- For each hospital: maintain a list of students, ordered by preference.
- For each hospital: maintain a pointer to student for next proposal.



**Bottom line.** Making a proposal takes  $O(1)$  time.

27

## Data representation: accepting/rejecting a proposal

**Student accepts/rejects a proposal.**

- Does student  $s$  prefer hospital  $h$  to hospital  $h'$ ?
- For each student, create **inverse** of preference list of hospitals.

pref[]	1 <sup>st</sup>	2 <sup>nd</sup>	3 <sup>rd</sup>	4 <sup>th</sup>	5 <sup>th</sup>	6 <sup>th</sup>	7 <sup>th</sup>	8 <sup>th</sup>
	8	3	7	1	4	5	6	2

↑

rank[]	1	2	3	4	5	6	7	8
	4 <sup>th</sup>	8 <sup>th</sup>	2 <sup>nd</sup>	5 <sup>th</sup>	6 <sup>th</sup>	7 <sup>th</sup>	3 <sup>rd</sup>	1 <sup>st</sup>

student prefers hospital 4 to 6  
since  $\text{rank}[4] < \text{rank}[6]$

```
for i = 1 to n
    rank[pref[i]] = i
```

**Bottom line.** After  $\Theta(n^2)$  preprocessing time (to create the  $n$  ranking arrays), it takes  $O(1)$  time to accept/reject a proposal.

28

## Stable matching: summary

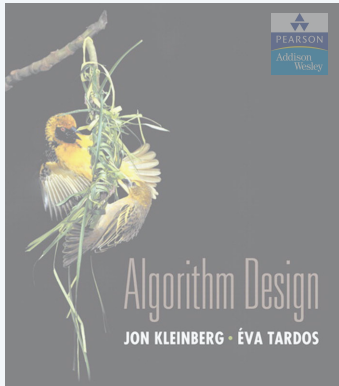
**Theorem.** Can implement Gale–Shapley to run in  $O(n^2)$  time.

**Pf.**

- $\Theta(n^2)$  preprocessing time to create the  $n$  ranking arrays.
- There are  $O(n^2)$  proposals; processing each proposal takes  $O(1)$  time. ■

**Theorem.** In the worst case, any algorithm to find a stable matching must query the hospital's preference list  $\Omega(n^2)$  times.

29



SECTION 2.4

## 2. ALGORITHM ANALYSIS

- ▶ *computational tractability*
- ▶ *asymptotic order of growth*
- ▶ *implementing Gale-Shapley*
- ▶ *survey of common running times*

## Constant time

**Constant time.** Running time is  $O(1)$ .

bounded by a constant,  
which does not depend on input size  $n$

### Examples.

- Conditional branch.
- Arithmetic/logic operation.
- Declare/initialize a variable.
- Follow a link in a linked list.
- Access element  $i$  in an array.
- Compare/exchange two elements in an array.
- ...

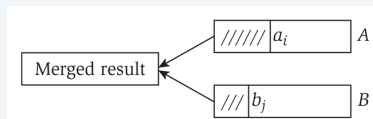
31

## Linear time

**Linear time.** Running time is  $O(n)$ .

**Merge two sorted lists.** Combine two sorted linked lists  $A = a_1, a_2, \dots, a_n$  and  $B = b_1, b_2, \dots, b_n$  into a sorted whole.

**$O(n)$  algorithm.** Merge in mergesort.



$i \leftarrow 1; j \leftarrow 1.$

**WHILE** (both lists are nonempty)

**IF** ( $a_i \leq b_j$ ) append  $a_i$  to output list and increment  $i$ .

**ELSE** append  $b_j$  to output list and increment  $j$ .

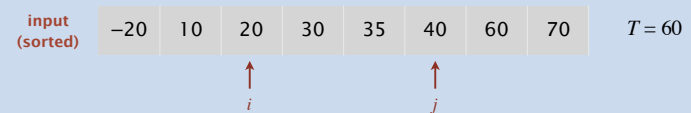
Append remaining elements from nonempty list to output list.

32

## TARGET SUM



**TARGET-SUM.** Given a sorted array of  $n$  distinct integers and an integer  $T$ , find two that sum to exactly  $T$ ?



33



## Logarithmic time

**Logarithmic time.** Running time is  $O(\log n)$ .

**Search in a sorted array.** Given a sorted array  $A$  of  $n$  distinct integers and an integer  $x$ , find index of  $x$  in array.

**$O(\log n)$  algorithm.** Binary search.

- Invariant: If  $x$  is in the array, then  $x$  is in  $A[lo \dots hi]$ .
- After  $k$  iterations of WHILE loop,  $(hi - lo + 1) \leq n / 2^k \Rightarrow k \leq 1 + \log_2 n$ .

$lo \leftarrow 1; hi \leftarrow n.$

WHILE ( $lo \leq hi$ )

$mid \leftarrow \lfloor (lo + hi) / 2 \rfloor.$

    IF ( $x < A[mid]$ )  $hi \leftarrow mid - 1.$

    ELSE IF ( $x > A[mid]$ )  $lo \leftarrow mid + 1.$

    ELSE RETURN  $mid.$

RETURN -1.



35

## Logarithmic time

$O(\log n)$

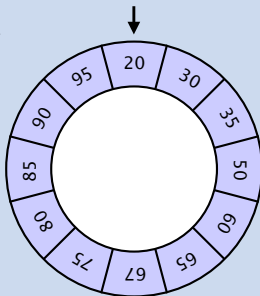
<https://www.facebook.com/pg/npcompleteteens>

## SEARCH IN A SORTED ROTATED ARRAY



**SEARCH-IN-SORTED-ROTATED-ARRAY.** Given a rotated sorted array of  $n$  distinct integers and an element  $x$ , determine if  $x$  is in the array.

sorted circular array



sorted rotated array

80	85	90	95	20	30	35	50	60	65	67	75
1	2	3	4	5	6	7	8	9	10	11	12

37

## Linearithmic time

**Linearithmic time.** Running time is  $O(n \log n)$ .

**Sorting.** Given an array of  $n$  elements, rearrange them in ascending order.

**$O(n \log n)$  algorithm.** Mergesort.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

39

## LARGEST EMPTY INTERVAL



**LARGEST-EMPTY-INTERVAL.** Given  $n$  timestamps  $x_1, \dots, x_n$  on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

40

## Quadratic time

**Quadratic time.** Running time is  $O(n^2)$ .

**Closest pair of points.** Given a list of  $n$  points in the plane  $(x_1, y_1), \dots, (x_n, y_n)$ , find the pair that is closest to each other.

**$O(n^2)$  algorithm.** Enumerate all pairs of points (with  $i < j$ ).

```
min ← ∞.  
FOR i = 1 TO n  
  FOR j = i + 1 TO n  
    d ← (xi - xj)2 + (yi - yj)2.  
    IF (d < min)  
      min ← d.
```

**Remark.**  $\Omega(n^2)$  seems inevitable, but this is just an illusion. [see §5.4]

42

## Cubic time

**Cubic time.** Running time is  $O(n^3)$ .

**3-SUM.** Given an array of  $n$  distinct integers, find three that sum to 0.

**$O(n^3)$  algorithm.** Enumerate all triples (with  $i < j < k$ ).

```
FOR i = 1 TO n  
  FOR j = i + 1 TO n  
    FOR k = j + 1 TO n  
      IF (ai + aj + ak = 0)  
        RETURN (ai, aj, ak).
```

**Remark.**  $\Omega(n^3)$  seems inevitable, but  $O(n^2)$  is not hard. [see next slide]

43

## 3-SUM



**3-SUM.** Given an array of  $n$  distinct integers, find three that sum to 0.

**$O(n^3)$  algorithm.** Try all triples.

**$O(n^2)$  algorithm.**

44

## Polynomial time

**Polynomial time.** Running time is  $O(n^k)$  for some constant  $k > 0$ .

**Independent set of size  $k$ .** Given a graph, find  $k$  nodes such that no two are joined by an edge.

$k$  is a constant

**$O(n^k)$  algorithm.** Enumerate all subsets of  $k$  nodes.

```
FOREACH subset  $S$  of  $k$  nodes:  
    Check whether  $S$  is an independent set.  
    IF ( $S$  is an independent set)  
        RETURN  $S$ .
```

- Check whether  $S$  is an independent set of size  $k$  takes  $O(k^2)$  time.
- Number of  $k$ -element subsets =  $\binom{n}{k} = \frac{n(n-1)(n-2) \times \dots \times (n-k+1)}{k(k-1)(k-2) \times \dots \times 1} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$ .

poly-time for  $k = 17$ ,  
but not practical

46

## Exponential time

**Exponential time.** Running time is  $O(2^{n^k})$  for some constant  $k > 0$ .

**Independent set.** Given a graph, find independent set of max cardinality.

**$O(n^2 2^n)$  algorithm.** Enumerate all subsets.

```
 $S^* \leftarrow \emptyset$ .  
FOREACH subset  $S$  of nodes:  
    Check whether  $S$  is an independent set.  
    IF ( $S$  is an independent set and  $|S| > |S^*|$ )  
         $S^* \leftarrow S$ .  
RETURN  $S^*$ .
```

47

## Analysis of algorithms: quiz 4



Which is an equivalent definition of exponential time?

- A.  $O(2^n)$
- B.  $O(2^{cn})$  for some constant  $c > 0$ .
- C. Both A and B.
- D. Neither A nor B.

48