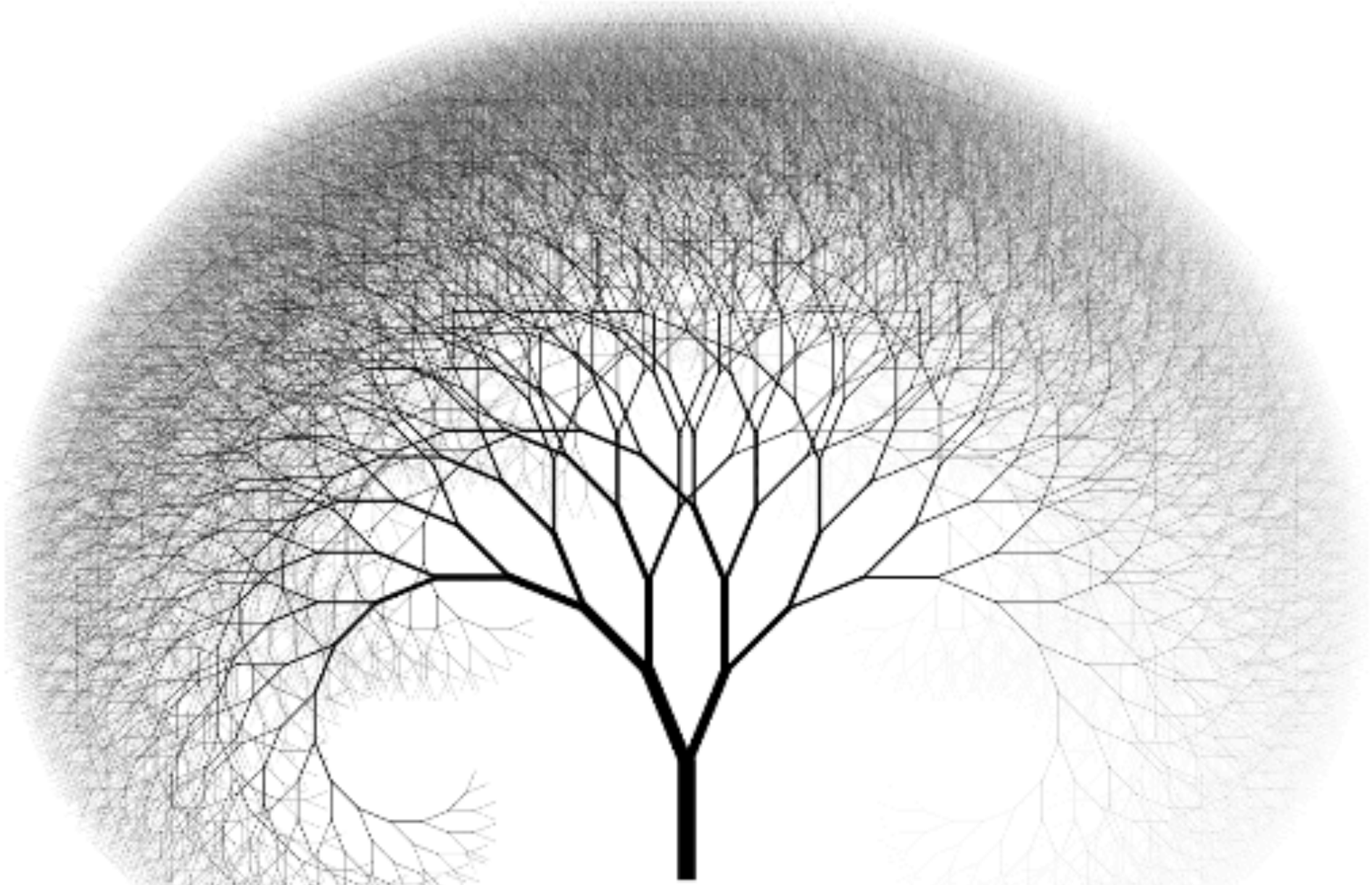


# 算法设计与分析

刘 安
苏州大学 计算机科学与技术学院
http://web.suda.edu.cn/anliu/

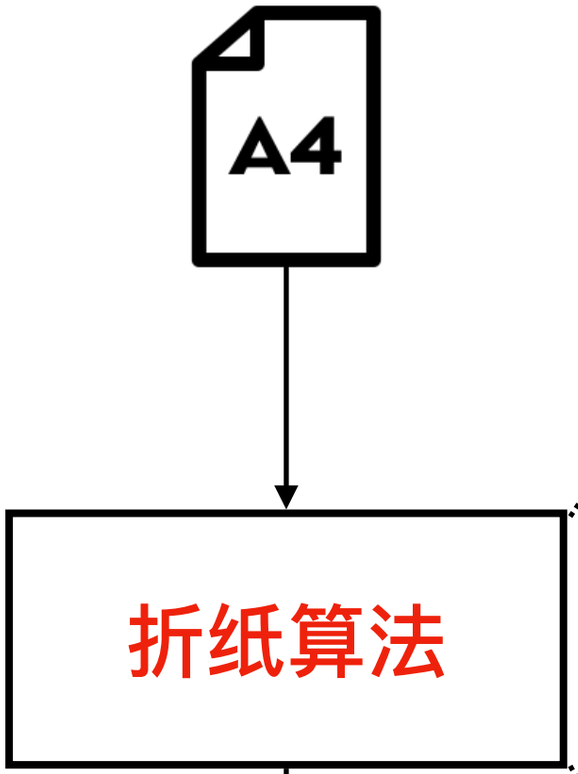
**致谢：本课件部分源于Kevin Wayne分享的课件**

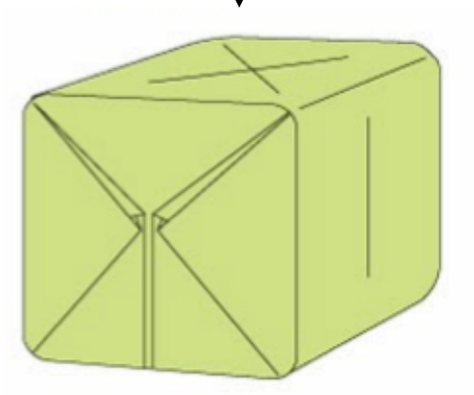
https://www.cs.princeton.edu/~wayne/kleinberg-tardos/

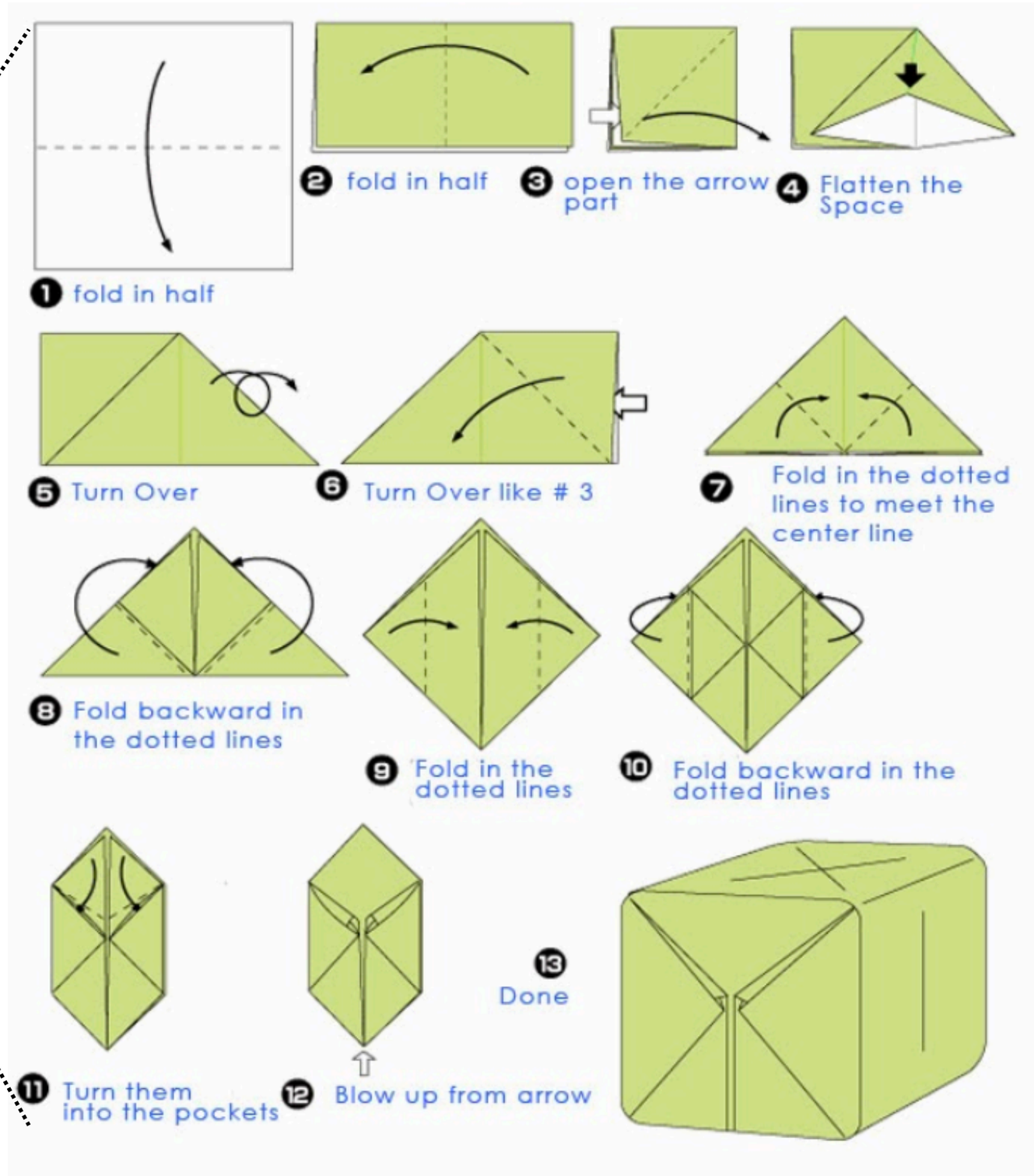
# 算法

- 算法是一个有限、确定、有效的过程，该过程将一些输入转换成一些输出
- 过程：一个具有逻辑顺序的特定步骤的序列
- 有限：必须在合理的时间内结束
- 确定：用清晰易懂的术语精确定义
- 有效：这些步骤必须能够实际执行

# 折纸算法


A4
折纸算法




1 fold in half
2 fold in half
3 open the arrow part
4 Flatten the Space
5 Turn Over
6 Turn Over like # 3
7 Fold in the dotted lines to meet the center line
8 Fold backward in the dotted lines
9 Fold in the dotted lines
10 Fold backward in the dotted lines
11 Turn them into the pockets
12 Blow up from arrow
13 Done

# 排序算法

输入 $\langle a_1, a_2, \cdots, a_n \rangle$

输入实例

| 31 | 41 | 59 | 26 | 41 | 58 |
|---|---|---|---|---|---|

**算法的正确性**

**对于每个输入实例，算法都以正确的输出结束**

**排序算法**

**排序算法**

```
# 一个神奇的排序算法
def a_sort(L):
    return [26, 31, 41, 41, 58, 59]
```

输出 $\langle a'_1, a'_2, \cdots, a'_n \rangle$

| 26 | 31 | 41 | 41 | 58 | 59 |
|---|---|---|---|---|---|

其中$a'_1, a'_2, \cdots, a'_n$是$a_1, a_2, \cdots, a_n$的一个排列，满足$a'_1 \leq a'_2 \leq \cdots \leq a'_n$

# 反面教材

## BeAMillionaireAndNeverPayTaxes()

---

**Get a million dollars.**

**If the tax man comes to your door and says, “You have never paid taxes!”**

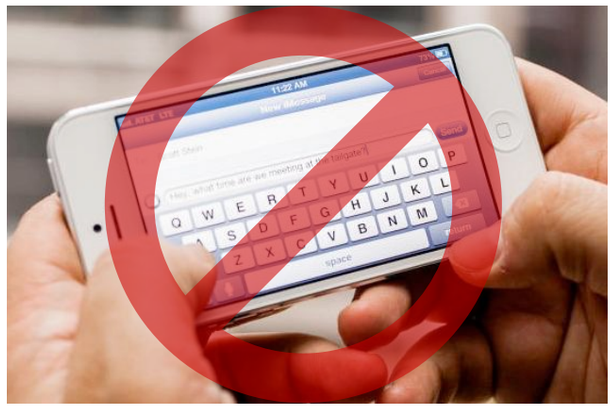
**Say “I forgot.”**

# 为什么学习算法

- 算法对计算机科学的所有分支都很重要
  - 计算机网络中的路由协议需要使用最短路径算法
  - 数据库中的查询依赖于搜索算法
  - 计算机图形学需要计算几何相关的算法
  - …
- 算法有助于在技术面试中取得成功
- …
- 算法很有趣！

# 课程相关说明

- 出席：5分
  - 每缺课1次，扣1分，扣完为止
- 平时作业：20分
  - 每少交1次，扣2分，每抄袭1次，扣2分，扣完为止
- 随堂或课后测试：15分
- 期末考试：60分

# 算法运行时间

- 给定一个列表，判断该列表中有无重复的元素，有重复元素，返回True，否则返回False

```
def duplicates1(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i != j and L[i] == L[j]:
                return True
    return False
```

```
In [27]: L1K = random.sample(range(10000), 1000)

In [28]: %timeit duplicates1(L1K)
79.8 ms ± 968 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

# 算法运行时间

- 给定一个列表，判断该列表中有无重复的元素，有重复元素，返回True，否则返回False

```
def duplicates2(L):
    n = len(L)
    for i in range(1, n):
        for j in range(i):
            if L[i] == L[j]:
                return True
    return False
```

```
def duplicates3(L):
    n = len(L)
    L.sort()
    for i in range(n-1):
        if L[i] == L[i+1]:
            return True
    return False
```

```
def duplicates4(L):
    s = set()
    for e in L:
        if e in s:
            return True
        s.add(e)
    return False
```

```
def duplicates5(L):
    return len(L) != len(set(L))
```

# 算法运行时间

**时间单位：毫秒**

| len(L) | duplicates1 | duplicates2 | duplicates3 | duplicates4 | duplicates5 |
|---|---|---|---|---|---|
| 1,000 | 79.8 | 29.6 | 2.87 | 0.078 | 0.016 |
| 10,000 | 8,250 | 3,190 | 37.4 | 0.922 | 0.312 |
| 100,000 | ? | ? | 480 | 12.8 | 5.97 |

- 不同算法的运行时间可能差异很大
- 某一算法的运行时间和诸多因素相关：机器、编程语言等
- 如何准确方便地描述算法的运行时间呢？

# 计算模型：随机访问机

- 随机访问机（random-access machine，RAM）
  - 基本操作：算术/逻辑运算、读写内存、数组索引、…
    - 假设基本操作花费的时间为常量
  - 每个内存单元可以存放一个w比特的整数
- 算法的运行时间：基本操作的数量
- 算法的空间需求：使用内存单元的数量
- 注意
  - 某些算法可能需要更精细的模型（比如2个大整数相乘）

# 算法运行时间

```python
def duplicates1(L):
    n = len(L) #01
    for i in range(n): #02
        for j in range(n): #03
            if i != j and L[i] == L[j]: #04
                return True #05
    return False #06
```

**最坏情况下，#基本操作** $= 1 + 2n^2 + 1 = 2n^2 + 2$

```python
def duplicates2(L):
    n = len(L) #01
    for i in range(1,n): #02
        for j in range(i): #03
            if L[i] == L[j]: #04
                return True #05
    return False #06
```

**最坏情况下，#基本操作=**$1 + \sum_{i=1}^{n} i + 1 = \frac{1}{2}n^2 + \frac{1}{2}n + 1$

# 算法运行时间

```python
def duplicates3(L):
    n = len(L) #01
    L.sort() #02
    for i in range(n - 1): #03
        if L[i] == L[i + 1]: #04
            return True #05
    return False #06
```

**最坏情况下，#基本操作** $= 1 + n\log n + (n-1) + 1 = n\log n + n + 1$

```python
def duplicates4(L):
    s = set() #01
    for e in L: #02
        if e in s: #03
            return True #04
        s.add(e) #05
    return False #06
```

**最坏情况下，#基本操作** $= 1 + 2n + 1 = 2n + 2$

```python
def duplicates5(L):
    return len(L) != len(set(L))
```

**最坏情况下，#基本操作** $= ?$

# 算法运行时间

**时间单位：毫秒**

| len(L) | duplicates1 | duplicates2 | duplicates3 | duplicates4 | duplicates5 |
|---|---|---|---|---|---|
| 1000 | 79.8 | 29.6 | 2.87 | 0.078 | 0.016 |
| 10000 | 8250 | 3190 | 37.4 | 0.922 | 0.312 |
| 100000 | ? | ? | 480 | 12.8 | 5.97 |

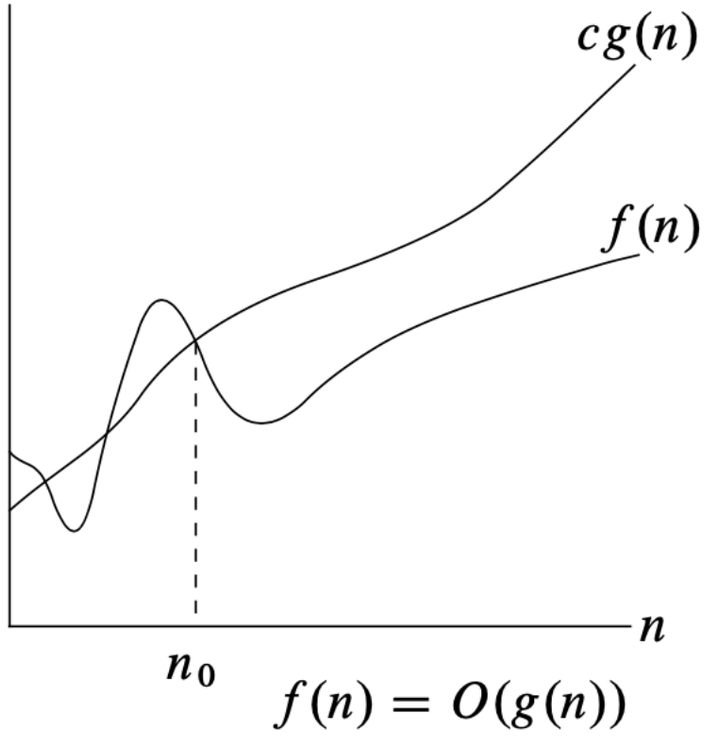
- duplicates1: $2n^2 + 2$
- duplicates2: $\frac{1}{2}n^2 + \frac{1}{2}n + 1$
- duplicates3: $n \log n + n + 1$
- duplicates4: $2n + 2$

# 渐近记号

- 使用基本操作的数量而不是实际运行时间来衡量算法运行时间
- 使用最坏情况分析
  - 给出了运行时间的上界
  - 仅仅考虑问题规模，而不关注具体输入，具有通用性
- 在分析基本操作数量时，渐近记号忽略
  - 常数因子：仍然和机器、编程语言等因素相关
  - 低阶项：输入规模很大时无关紧要
- 对长度为$n$的数组，duplicates2最多执行$\frac{1}{2}n^2+\frac{1}{2}n+1$次基本操作
  - 忽略低阶项：$\frac{1}{2}n^2+\frac{1}{2}n+1 \Rightarrow \frac{1}{2}n^2$
  - 忽略常数因子：$\frac{1}{2}n^2 \Rightarrow n^2$

# O记号

- $O(g(n))$ = {$f(n)$: 存在正常量$c$和$n_0$，使得对所有$n \geq n_0$，有 $0 \leq f(n) \leq c \cdot g(n)$}
- $f(n) = O(g(n))$表示函数$f(n)$是函数集合$O(g(n))$的成员
- 如果 $f(n) = 32n^2 + 17n + 1$，那么 $f(n) = O(n^2)$
  - 令$c = 50$，$n_0 = 1$，对于所有的$n \geq n_0 = 1$，
    $f(n) = 32n^2 + 17n + 1 \leq 50n^2 = c \cdot n^2$




- 如何知道$c = 50$，$n_0 = 1$呢？
  - $g(n) = n^2 \Rightarrow c \cdot g(n) = c \cdot n^2$
  - $f(n) = 32n^2 + 17n + 1 \leq 32n^2 + 17n^2 + n^2 = 50n^2$ （当$n \geq 1$时）
  - 所以令$c = 50$，$n_0 = 1$

# O记号

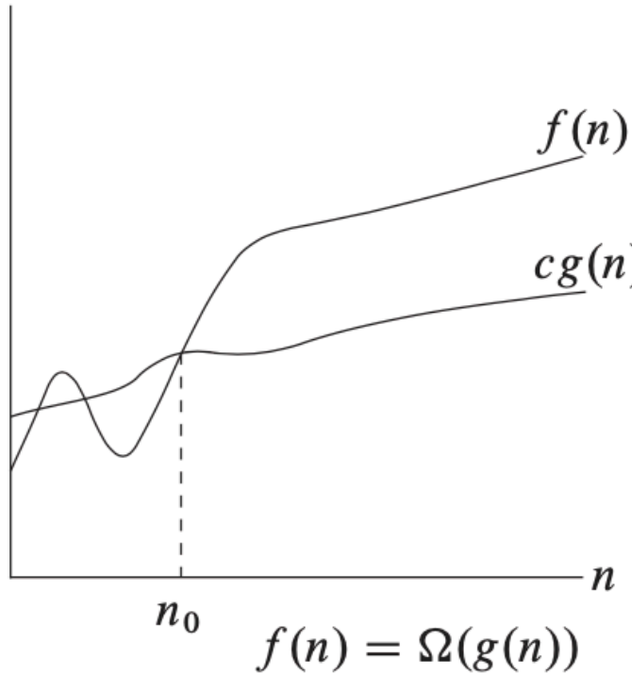
- $O(g(n)) = \{f(n)$: 存在正常量$c$和$n_0$，使得对所有$n \geq n_0$，有$0 \leq f(n) \leq c \cdot g(n)\}$
- $f(n) = O(g(n))$表示函数$f(n)$是函数集合$O(g(n))$的成员
- 如果 $f(n) = 32n^2 + 17n + 1$，那么 $f(n) = O(n^2)$
- $f(n)$不是$O(n)$，也不是$O(\log n)$
  - 下面通过反证法来证明$f(n)$不是$O(n)$
  - 假设存在$c$和$n_0$，使得对所有$n \geq n_0$，有$32n^2 + 17n + 1 \leq c \cdot n$
  - 两边同除以$n$，有$32n + 17 + \frac{1}{n} \leq c$，即$32n \leq c - 17 - \frac{1}{n} \leq c$
  - 当$32n > c$时，上面的不等式不成立，所以$f(n)$不是$O(n)$

# $O$记号的性质

- 自反. $f = O(f)$
- 常量. 如果$f = O(g)$并且$c > 0$，那么$c \cdot f = O(g)$
- 乘积. 如果$f_1 = O(g_1)$并且$f_2 = O(g_2)$，那么$f_1 f_2 = O(g_1 g_2)$
  - 存在正常量$c_1$和$n_1$，使得对于所有$n \geq n_1$，有$0 \leq f_1(n) \leq c_1 \cdot g_1(n)$
  - 存在正常量$c_2$和$n_2$，使得对于所有$n \geq n_2$，有$0 \leq f_2(n) \leq c_2 \cdot g_2(n)$
  - 所以，对于所有的$n \geq \max\{n_1, n_2\}$，有
    $0 \leq f_1(n) \cdot f_2(n) \leq c_1 \cdot c_2 \cdot g_1(n) \cdot g_2(n)$
- 和. 如果$f_1 = O(g_1)$并且$f_2 = O(g_2)$，那么$f_1 + f_2 = O(\max\{g_1, g_2\})$
- 传递. 如果$f = O(g)$并且$g = O(h)$，那么$f = O(h)$

# Ω记号

- $\Omega(g(n)) = \{f(n)$: 存在正常量$c$和$n_0$，使得对所有$n \geq n_0$，有$0 \leq c \cdot g(n) \leq f(n)\}$
- $f(n) = \Omega(g(n))$表示函数$f(n)$是函数集合$\Omega(g(n))$的成员
- 如果 $f(n) = 2n^3 - 7n + 1$，那么 $f(n) = \Omega(n^3)$
  - 令$c = 1$，$n_0 = 3$，对于所有的$n \geq n_0 = 3$，
    $f(n) = 2n^3 - 7n + 1 = n^3 + (n^3 - 7n) + 1 \geq n^3 + 1 \geq n^3$



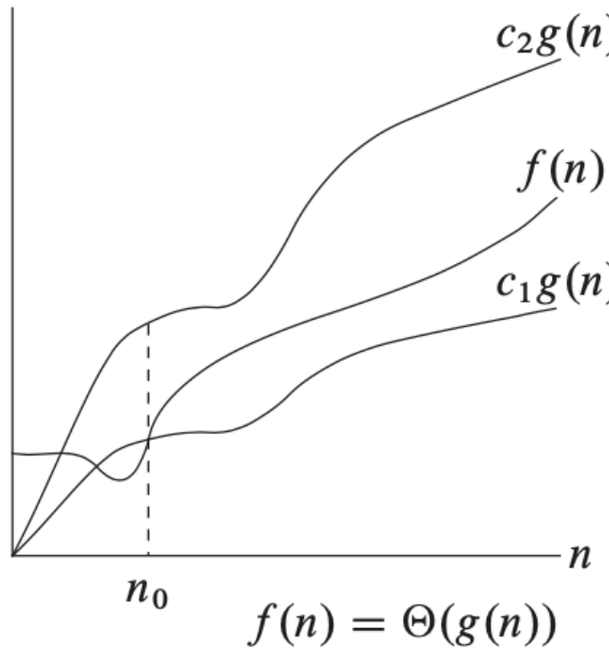

- 如何知道$c = 1$，$n_0 = 3$呢？
  - $g(n) = n^3 \Rightarrow c \cdot g(n) = c \cdot n^3$
  - $f(n) = 2n^3 - 7n + 1 = n^3 + (n^3 - 7n) + 1$
  - 当$n^3 - 7n > 0$时，即$n > 3$时，可以将$f(n)$进一步缩小为$n^3$
  - 所以令$c = 1$，$n_0 = 3$

# Ω记号

- $\Omega(g(n)) = \{f(n)$: 存在正常量$c$和$n_0$，使得对所有$n \geq n_0$，有$0 \leq c \cdot g(n) \leq f(n)\}$
- $f(n) = \Omega(g(n))$表示函数$f(n)$是函数集合$\Omega(g(n))$的成员
- 如果 $f(n) = 2n^3 - 7n + 1$，那么$f(n) \neq \Omega(n^4)$
  - 假设存在正常量$c$和$n_0$，使得对所有$n \geq n_0$，有$2n^3 - 7n + 1 \geq c \cdot n^4$
    - 看能否推出矛盾，即$2n^3 - 7n + 1 < c \cdot n^4$
  - 显然，$2n^3 - 7n + 1 \leq 2n^3 + 1 \leq 2n^3 + n^3 = 3n^3$ （当$n \geq 1$时）
  - 因为目标是$n^4$，所以$2n^3 - 7n + 1 \leq 3n^3 \leq c \cdot \frac{3}{c} n^3 \dashrightarrow c \cdot n^4$
  - 当$\frac{3}{c} < n$时，$c \cdot \frac{3}{c} n^3 < c \cdot n^4$
  - 综合上述考虑，令$n > \max\{1, \frac{3}{c}\}$，可以得到$2n^3 - 7n + 1 < c \cdot n^4$，矛盾！

# Θ记号

- $\Theta(g(n)) = \{f(n)$: 存在正常量$c_1, c_2$和$n_0$，使得对所有$n \geq n_0$，有$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$
- $f(n) = \Theta(g(n))$表示函数$f(n)$是函数集合$\Theta(g(n))$的成员
- $f(n) = 32n^2 + 17n + 1$
  - $f(n) = \Theta(n^2)$
  - $f(n) \neq \Theta(n)$，$f(n) \neq \Theta(n^3)$




- 定理：对任意两个函数$f(n)$和$g(n)$，$f(n) = \Theta(g(n))$当且仅当$f(n) = O(g(n))$并且$f(n) = \Omega(g(n))$
- 如果$f(n) = \Theta(g(n))$，$g(n)$称为$f(n)$的渐近紧确界
  - 如果$f(n) = O(g(n))$，$g(n)$称为$f(n)$的渐近上界
  - 如果$f(n) = \Omega(g(n))$，$g(n)$称为$f(n)$的渐近下界

# 思考题

😇 假设$T(n) = \frac{1}{2}n^2 + 3n$，下面哪些说法是正确的？

A. $T(n) = O(n)$

B. $T(n) = \Omega(n)$

C. $T(n) = \Theta(n^2)$

D. $T(n) = O(n^3)$

# 渐近记号与极限

- 如果 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ ，其中$c$是一个正常量， 那么 $f(n) = \Theta(g(n))$
  - 根据极限定义，对任意 $\epsilon > 0$, 存在 $n_0$ 使得对于所有$n \geq n_0$，有$c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon$
  - 令 $\epsilon = \frac{1}{2}c > 0$，两边同乘以 $g(n)$ ，有 $\frac{1}{2}c \cdot g(n) \leq f(n) \leq \frac{3}{2}c \cdot g(n)$
  - 令$c_1 = 1/2 \cdot c$， $c_2 = 3/2 \cdot c$，根据$\Theta$定义，有$f(n) = \Theta(g(n))$
- 如果 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$， 那么$f(n) = O(g(n))$，$f(n) \neq \Omega(g(n))$
  - 此时，也称$f(n) = o(g(n))$，其中$o$记号表示一个非渐近紧确的上界
  - 比如，$2n = o(n^2)$，但$2n^2 \neq o(n^2)$
- 如果 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$， 那么$f(n) = \Omega(g(n))$，$f(n) \neq O(g(n))$
  - 此时，也称$f(n) = \omega(g(n))$，其中$\omega$记号表示一个非渐近紧确的下界
  - 比如，$n^2/2 = \omega(n)$，但$n^2/2 \neq \omega(n^2)$

# 常见函数的渐近记号

- 多项式函数： Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ with $a_d > 0$. Then $f(n)$ is $\Theta(n^d)$.
  - $\lim_{n \to \infty} \frac{a_0 + a_1 n + \cdots + a_d n^d}{n^d} = a_d > 0$
- 对数函数： $\log_a n$ is $\Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.
  - $\frac{log_a n}{\log_b n} = \frac{1}{\log_b a}$
- 对数函数与多项式函数. $\log_a n$ is $O(n^d)$ for every $a > 1$ and every $d > 0$.
  - $\lim_{n \to \infty} \frac{\log_a n}{n^d} = 0$
- 指数函数与多项式函数. $n^d$ is $O(r^n)$ for every $r > 1$ and every $d > 0$.
  - $\lim_{n \to \infty} \frac{n^d}{r^n} = 0$

# 具有不同渐近界的算法的运行时间

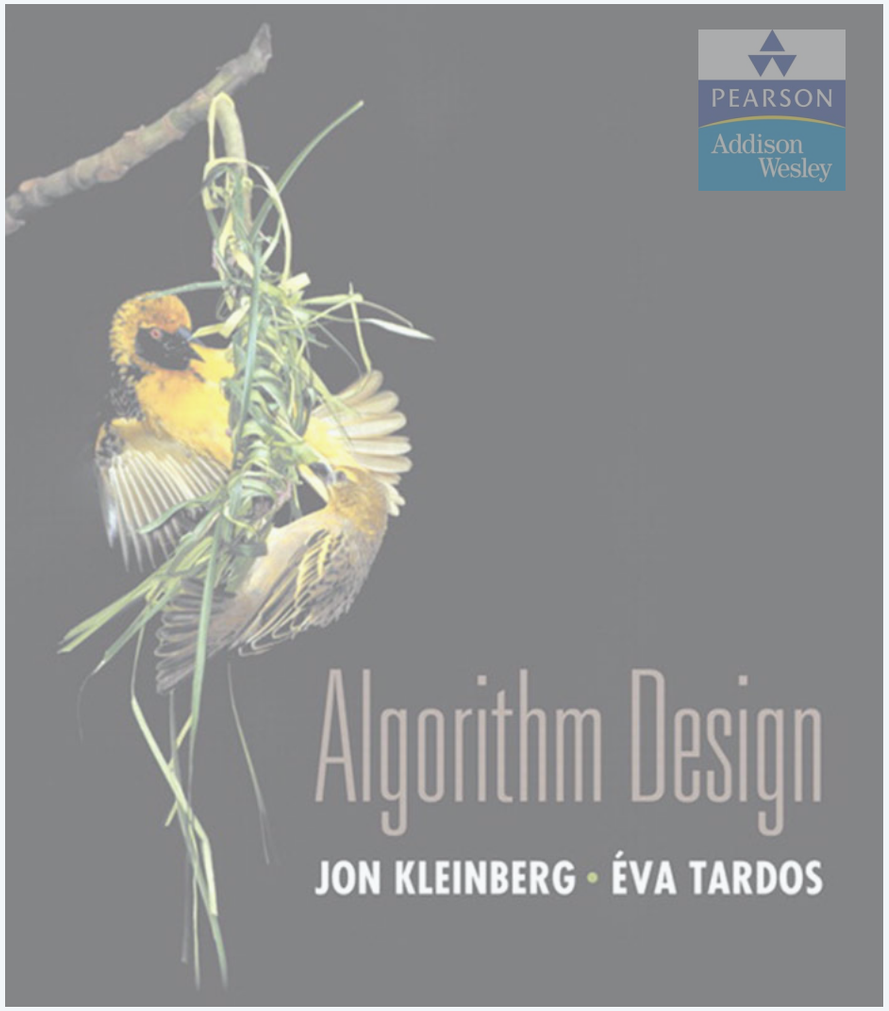
**Table 2.1** The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 4 sec |
| $n = 30$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 18 min | $10^{25}$ years |
| $n = 50$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 11 min | 36 years | very long |
| $n = 100$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | $< 1$ sec | $< 1$ sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | $< 1$ sec | $< 1$ sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | $< 1$ sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |

| 响应时间 | 输入规模 | 能接受的渐近运行时间 |
|---|---|---|
| 1秒 | 1,000,000 | $O(n)$ |
| | 100,000 | $O(n \log n)$ |
| | 10,000 | $O(n^2)$ |

# 等式中的渐近记号

- 等式的右边只有渐近记号，比如$n = O(n^2)$
  - 等号实际上是集合的成员关系，即$n \in O(n^2)$
- 等式的右边包含渐近记号，比如$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - $\Theta(n)$代表一个匿名函数$f(n)$，其中$f(n) \in \Theta(n)$
  - 用于隐藏无关紧要的细节
- 等式的左边包含渐近记号，比如$2n^2 + \Theta(n) = \Theta(n^2)$
  - 无论怎样选择等号左边的匿名函数$f(n) \in \Theta(n)$，总有一种办法来选择等号右边的匿名函数$g(n) \in \Theta(n^2)$，使得等式成立，即$2n^2 + f(n) = g(n)$




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.

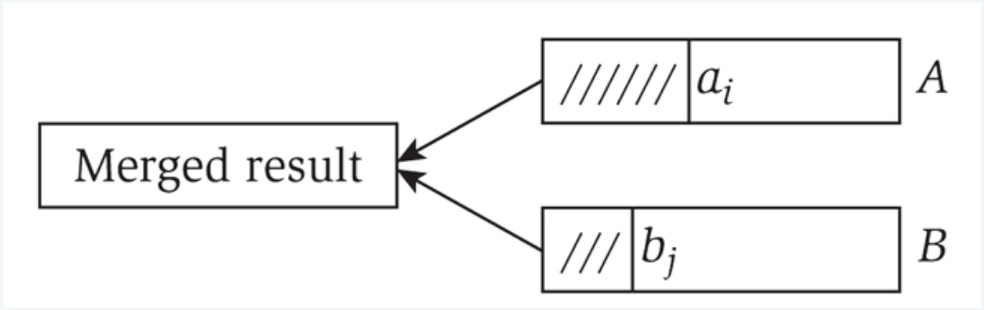
- Conditional branch.
- Arithmetic/logic operation.
- Declare/initialize a variable.
- Follow a link in a linked list.
- Access element $i$ in an array.
- Compare/exchange two elements in an array.
- …

# Linear time

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

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

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




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

WHILE (both lists are nonempty)

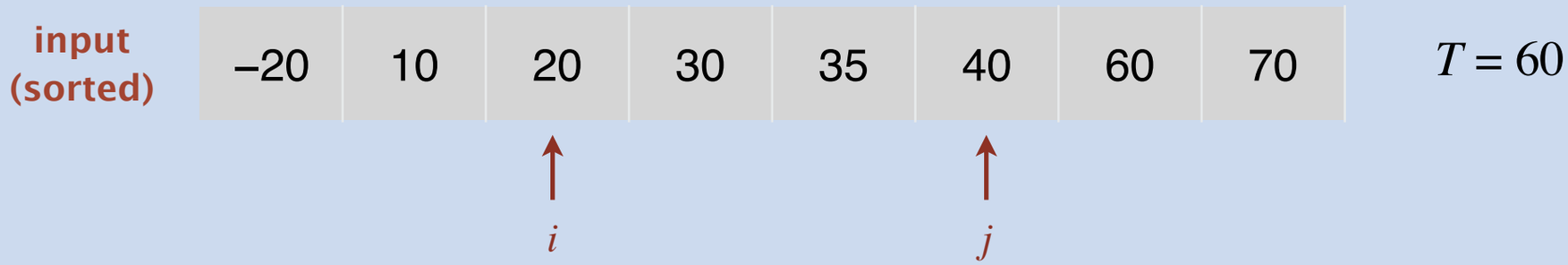
  IF ($a_i \leq b_j$) append $a_i$ to output list and increment $i$.

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

Append remaining elements from nonempty list to output list.

# TARGET SUM

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

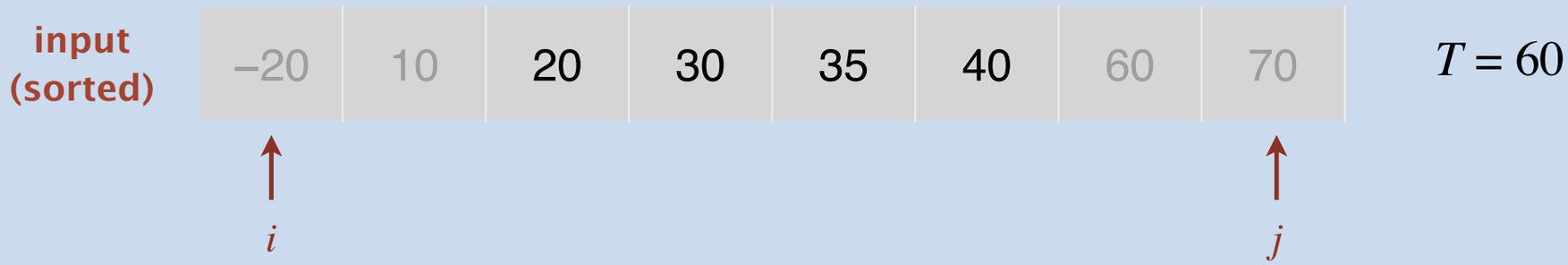


# TARGET SUM

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

$O(n^2)$ algorithm. Try all pairs.

$O(n)$ algorithm. Exploit sorted order.




Invariant. No element to the left of $i$ or right of $j$ in pair that sums to $T$.

# 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.

remaining elements

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

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

WHILE $(lo \le hi)$

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

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

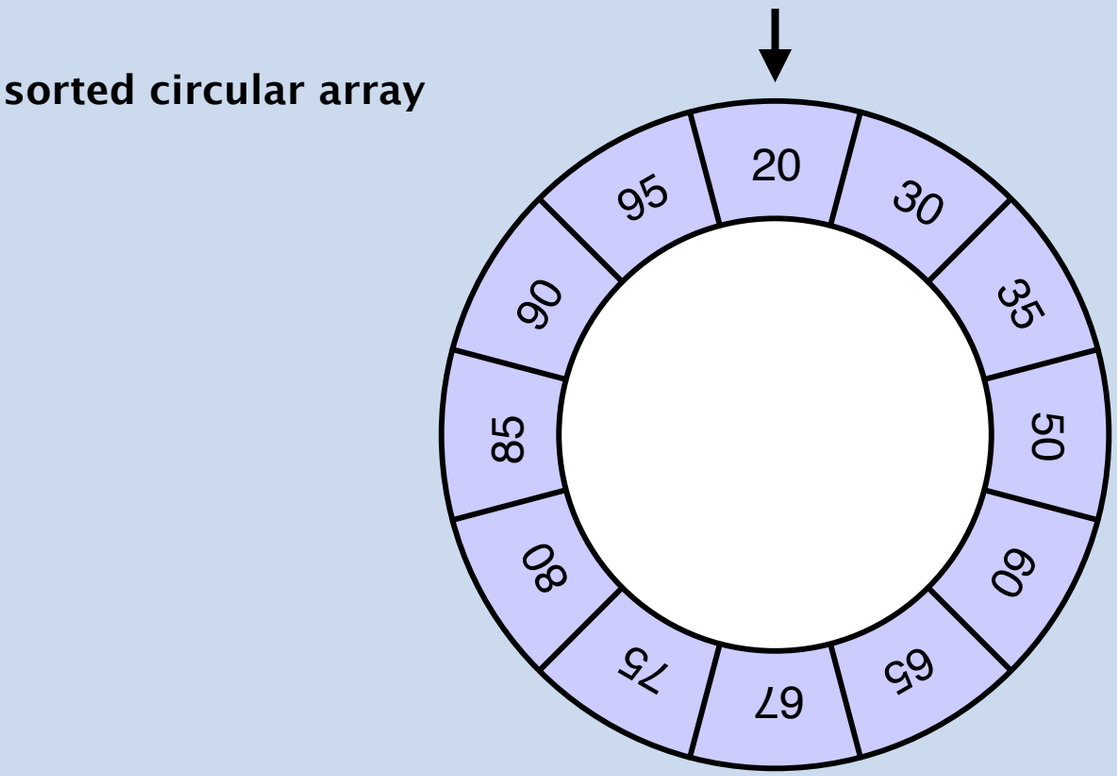
  ELSE IF $(x > A[mid])$ $lo \leftarrow mid + 1.$

  ELSE RETURN $mid.$

RETURN $-1.$

# 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 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 |

# 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.

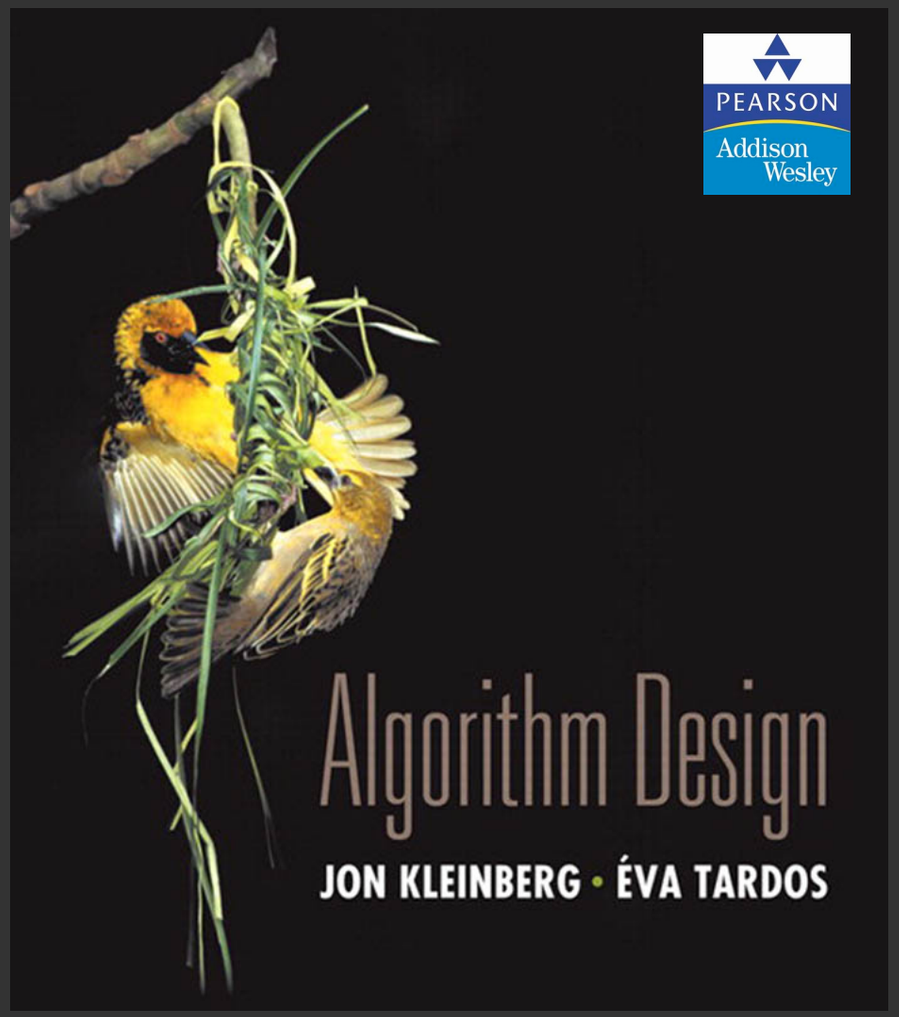
$O(\log n)$ algorithm.

- Find index $k$ of smallest element.
- Binary search for $x$ in either $A[1 \,..\, k\text{–}1]$ or $A[k \,..\, n]$.

**find index of smallest element**

```
lo ← 1; hi ← n.
IF (A[lo] ≤ A[hi]) RETURN 0          ⟵ sorted
WHILE (lo + 2 ≤ hi)                  ⟵ at least 3 elements
    mid ← ⌊(lo + hi) / 2⌋.
    IF      (A[mid] < A[hi])  hi ← mid.
    ELSE IF (A[mid] > A[hi])  lo ← mid.
RETURN hi
```

loop invariant
$A[lo] > A[hi]$




# 2. Algorithm Analysis

- *binary search demo*
- *sorted rotated array demo*



http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 2021/9/18 上午8:48

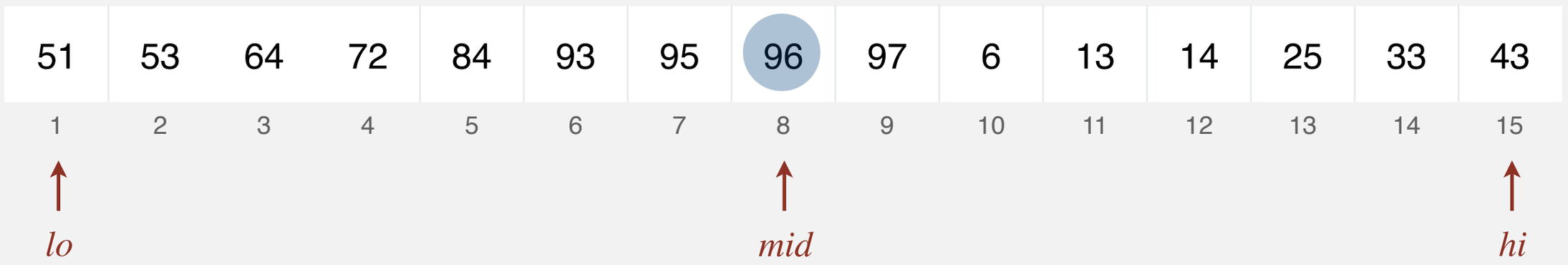
# Sorted rotated array demo

Goal. Given a rotated sorted array of $n$ distinct integers, find index of smallest element.

Invariant. $A[lo] > A[hi]$.

Binary search. Compare middle entry $A[mid]$ to last entry $A[hi]$.

- Less, go left.
- Bigger, go right.



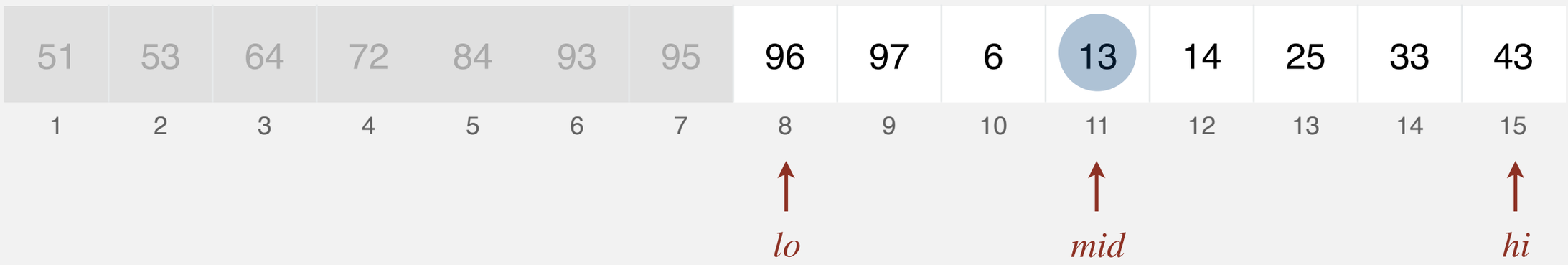
# Sorted rotated array demo

Goal. Given a rotated sorted array of $n$ distinct integers, find index of smallest element.

Invariant. $A[lo] > A[hi]$.

Binary search. Compare middle entry $A[mid]$ to last entry $A[hi]$.

- Less, go left.
- Bigger, go right.



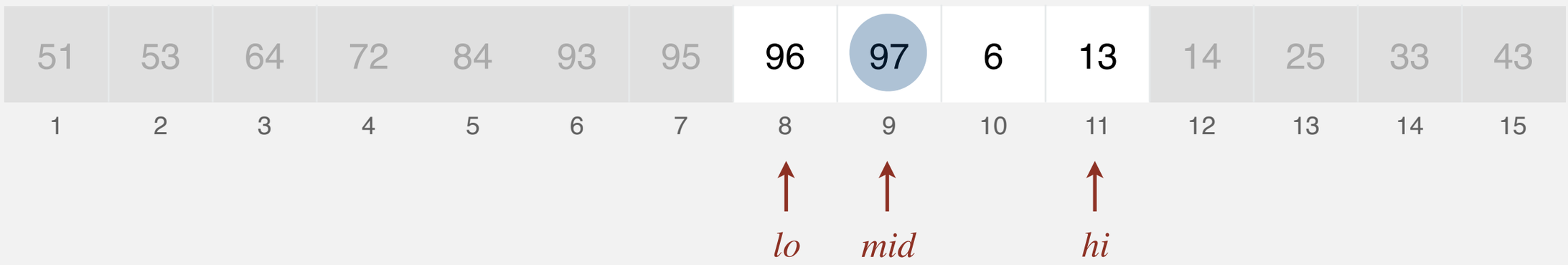
# Sorted rotated array demo

Goal. Given a rotated sorted array of $n$ distinct integers, find index of smallest element.

Invariant. $A[lo] > A[hi]$.

Binary search. Compare middle entry $A[mid]$ to last entry $A[hi]$.

- Less, go left.
- Bigger, go right.



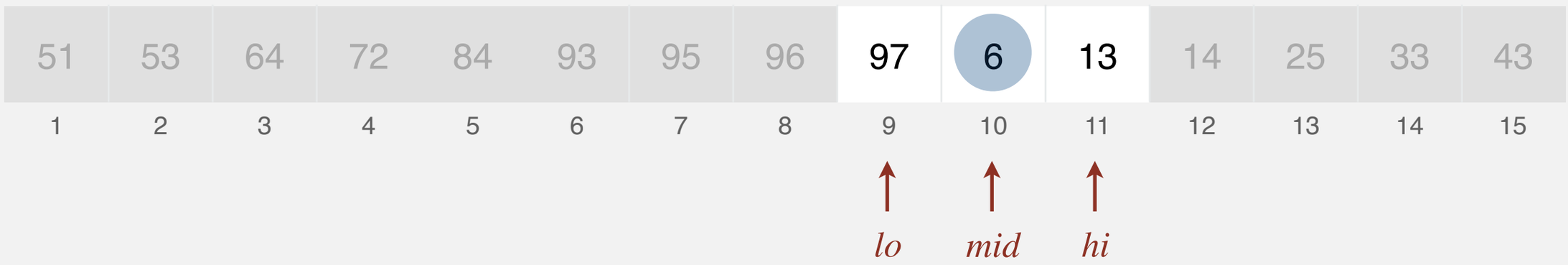
# Sorted rotated array demo

Goal. Given a rotated sorted array of $n$ distinct integers, find index of smallest element.

Invariant. $A[lo] > A[hi]$.

Binary search. Compare middle entry $A[mid]$ to last entry $A[hi]$.

- Less, go left.
- Bigger, go right.



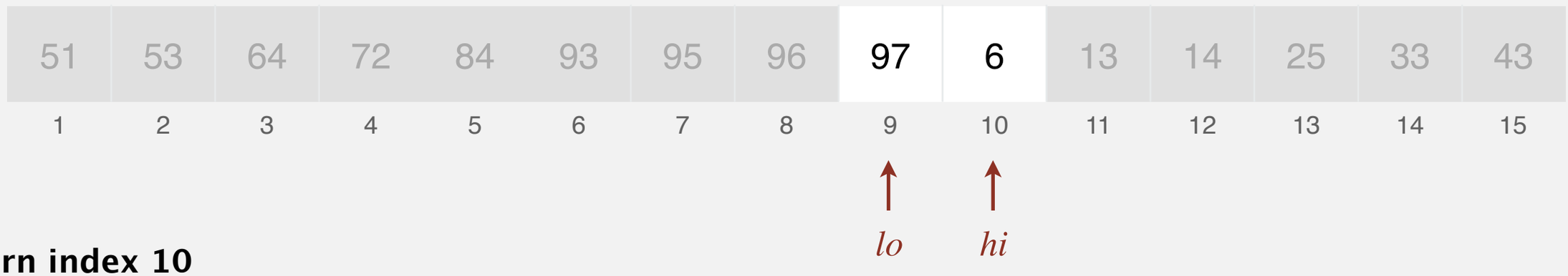
# Sorted rotated array demo

Goal. Given a rotated sorted array of $n$ distinct integers, find index of smallest element.

Invariant. $A[lo] > A[hi]$.

Binary search. Compare middle entry $A[mid]$ to last entry $A[hi]$.

- Less, go left.
- Bigger, go right.




**return index 10**

# 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 | G | M | R | E | O | R | S | T | E | X | A | M | P | L | E |
| E | E | G | M | O | R | R | S | 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 | 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 |

# Largest Empty interval

Largest-Empty-Interval. Given $n$ timestamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

# LARGEST EMPTY INTERVAL

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

$O(n \log n)$ algorithm.

- Sort the array $a$.
- Scan the sorted list in order, identifying the maximum gap between successive timestamps.

# Quadratic time

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

Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, 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 ← (x_i − x_j)^2 + (y_i − y_j)^2.
        IF (d < min)
            min ← d.
```

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

# 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 (a_i + a_j + a_k = 0)
                RETURN (a_i, a_j, a_k).
```

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

# 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.

# 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.

- Sort the array $a$.
- For each integer $a_i$ : solve TARGET-SUM on the array containing all elements except $a_i$ with the target sum $T = -a_i$.

Best-known algorithm. $O(n^2 / (\log n / \log \log n))$.

Conjecture(推测). No $O(n^{2-\varepsilon})$ algorithm for any $\varepsilon > 0$.

# 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

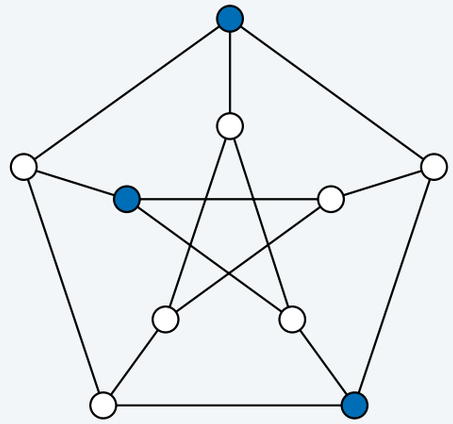
$O(n^k)$ algorithm. Enumerate all subsets of $k$ nodes.

FOREACH subset $S$ of $k$ nodes:

  Check whether $S$ is an independent set.

  IF ($S$ is an independent set)

    RETURN $S$.



independent set of size 3

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

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

# 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 size.

$O(n^2\, 2^n)$ algorithm. Enumerate all subsets of $n$ elements.

$S^* \leftarrow \emptyset$.

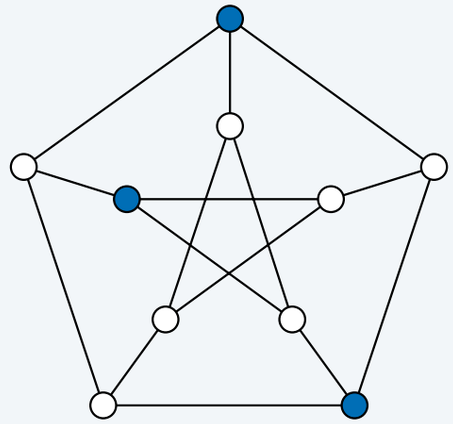
FOREACH subset $S$ of $n$ nodes:

  Check whether $S$ is an independent set.

  IF ($S$ is an independent set and $|S| > |S^*|$)

    $S^* \leftarrow S$.

RETURN $S^*$.



independent set of max size

# Exponential time

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

Euclidean TSP. Given $n$ points in the plane, find a tour of minimum length.

$O(n \times n!)$ algorithm. Enumerate all permutations of length $n$.

$\pi^* \leftarrow \emptyset$.

FOREACH permutation $\pi$ of $n$ points:

  Compute length of tour corresponding to $\pi$.

  IF (length($\pi$) < length($\pi^*$))

    $\pi^* \leftarrow \pi$.

RETURN $\pi^*$.

for simplicity, we'll assume Euclidean distances are rounded to nearest integer (to avoid issues with infinite precision)