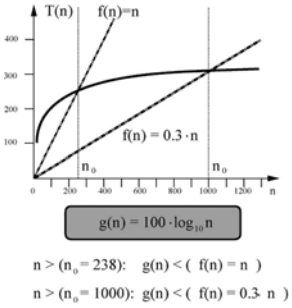
$g(n)$ is $O(f(n))$, or $g(n) = O(f(n))$

Function $g(n)$ is "Big-Oh" of $f(n)$ if, starting from some $n > n_0$, there always exist a function $c \cdot f(n)$ that grows faster than the function $g(n)$



"Big-Oh" $O(\ldots)$: Informal Meaning

- g is $O(f)$ means that the order of time complexity of the function g is ***asymptotically* less than** or **equal to** the order of time complexity of the function f
  - Asymptotical behaviour ↔ only for the large values of n
  - Two functions are of the same order when they each are "Big-Oh" of the other: $f = O(g)$ AND $g = O(f)$
  - This property is called "Big-Theta": $g = \Theta(f)$

"Big-Oh" $O(\ldots)$: Informal Meaning

g is $O(f)$ means that the algorithm with time complexity g runs (for large n) at least as fast, within a constant factor, as the algorithm with time complexity f

$$\lim_{n\to\infty} \left\{ g(n) \Big/ f(n) \right\} = c$$

Big-Omega

- The function $g(n)$ is $\Omega(f(n))$ iff there exist a real positive constant $c > 0$ and a positive integer n_0 such that $g(n) \geq cf(n)$ for all $n \geq n_0$
  - Big Omega is just opposite to Big Oh
  - It generalises the concept of "lower bound" (≥) in the same way as Big Oh generalises the concept of "upper bound" (≤)
  - If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$

Big-Theta

- The function $g(n)$ is $\Theta(f(n))$ iff there exist two real positive constants $c_1 > 0$ and $c_2 > 0$ and a positive integer n_0 such that:

  $c_1 f(n) \geq g(n) \geq c_2 f(n)$ for all $n \geq n_0$
  - Whenever two functions, f and g, are of the same order, $g(n)$ is $\Theta(f(n))$, they are each Big-Oh of the other: $g(n)$ is $O(f(n))$ AND $f(n)$ is $O(g(n))$

Upper bounds of complexity

"Big-Oh" specifies an **upper bound of complexity** so that the following (and like) relationships hold:

$1 = O(\log n) = O(n) = O(n \log n) = \ldots$

$\log n = O(n) = O(n \log n) = O(n^\alpha);\ \alpha > 1 = \ldots$

$n = O(n \log n) = O(n^\alpha);\ \alpha > 1 = O(2^n) = \ldots$

$n \log n = O(n^\alpha);\ \alpha > 1 = O(n^k);\ k > \alpha = \ldots$

$n^k = O(n^\alpha);\ \alpha > k = O(2^n) = \ldots$

Time complexity growth

$f(n)$	Number of data items processed per:			
	1 minute	1 day	1 year	1 century
n	**10**	**14,400**	**$5.26 \cdot 10^6$**	**$5.26 \cdot 10^8$**
$n \log n$	10	3,997	883,895	$6.72 \cdot 10^7$
$n^{1.5}$	10	1,275	65,128	$1.40 \cdot 10^6$
n^2	10	379	7,252	72,522
n^3	10	112	807	3,746
2^n	**10**	**20**	**29**	**35**

7

Beware exponential complexity

☺ If a linear, $O(n)$, algorithm processes **10** items per minute, then it can process **14,400** items per day, **5,260,000** items per year, and **526,000,000** items per century.

☻ If an exponential, $O(2^n)$, algorithm processes **10** items per minute, then it can process only **20** items per day and **35** items per century...

8

Ascending order of complexity

$1 \leftarrow \log\log n \leftarrow \log n \leftarrow n \leftarrow n \log n$
$\leftarrow n^{\alpha};\ 1<\alpha<2 \leftarrow n^2 \leftarrow n^3 \leftarrow n^m;\ m>3 \leftarrow 2^n \ldots$

Questions:

- Where is the place of $n^3 \log n$?
- Where is the place of $n^{2.79}$?

9

Answer:

$1 \leftarrow \log\log n \leftarrow \log n \leftarrow n \leftarrow n \log n$
$\leftarrow n^{\alpha};\ 1<\alpha<2 \leftarrow n^2 \leftarrow \mathbf{n^{2.79}} \leftarrow n^3 \leftarrow$
$n^3 \log n \leftarrow n^m;\ m>3 \leftarrow 2^n \ldots$

10

- The exact running time function is not important, since it can be multiplied by any arbitrary positive constant.
- Two functions are compared only ***asymtotically***, for large n, and not near the origin
  - If the constants involved are very large, then the asymptotical behaviour is of no practical interest!

11

Example

- Let algorithms **A** and **B** have running times $T_A(n) = 20n$ ms and $T_B(n) = 0.1n \log_2 n$ ms
- In the "Big-Oh"sense, **A** is better than **B**…
- But: on which data volume can **A** outperform **B**?

  $T_A(n) < T_B(n)$ if $20n < 0.1n \log_2 n$, or
  $\log_2 n > 200$, that is, when $n > 2^{200} \approx 10^{60}$!
- Thus, in all practical cases **B** is better than **A**…

12

Example

Let algorithms **A** and **B** have running times $T_A(n) = 20n$ ms and $T_B(n) = 0.1n^2$ ms

- In the "Big-Oh"sense, **A** is better than **B**…
- But: on which data volumes **A** outperforms **B**?

  $T_A(n) < T_B(n)$ if $20n < 0.1n^2$, or ***n > 200***
- Thus **A** is better than **B** in most practical cases except for $n < 200$ when **B** becomes faster…

13

Examples

Running time T(*n*)	**Complexity O(*n*)**
$n^2 + 100\,n + 1$	$O(n^2)$
$0.001n^3 + n^2 + 1$	$O(n^3)$
$23\,n$	$O(n)$
2^{3n}	$O(8^n)$ as $2^{3n} \equiv (2^3)^n$
2^{3+n}	$O(2^n)$ as $2^{3+n} \equiv 2^3 \cdot 2^n$
$2 \cdot 3^n$	$O(3^n)$

14

Examples

Running time T(*n*)	**Complexity O(*n*)**
$0.0001\,n + 10000$	$O(n)$
$100000\,n + 10000$	$O(n)$
$0.0001\,n^2 + 10000\,n$	$O(n^2)$
$100000\,n^2 + 10000\,n$	$O(n^2)$
$30 \log_{20}(23n)$ **actually NOT that hard…**	$O(\log n)$ as $\log_c(ab) = \log_c a + \log_c b$

15

Example

Assume that we have an algorithm with a running time

$$T(n) = \sqrt{n}$$

that can process x input items on your computer.

What input size it can process if the computer is 100 times faster?

$$\sqrt{n_{new}} = 100\ \sqrt{x}$$
$$n_{new} = 10^4\, x$$

16

Worst-Case Performance

- **Upper bounds**: simple to obtain
- **Lower bounds**: ***a difficult matter***...
- Worst case data may be unlikely to be met in practice
- Unknown "Big-Oh" constants c and n_0 may not be small
- Inputs in practice lead to much lower running times
- ***Example:*** the most popular fast sorting algorithm, **QuickSort**, has $O(n^2)$ running time in the worst case but in practice the time is $O(n \log n)$

17

Average-Case Performance

- Estimate average time for each operation
- Estimate frequencies of operations
- May be a difficult challenge... (take **COMPSCI.320** for details)
- May be no natural "average" input at all
- May be hard to estimate (average time for each operation depends on data)

18

Recurrence Relations

Divide-and-conquer principle:

- to divide a large problem into smaller ones and recursively solve each subproblem, then
- to combine solutions of the subproblems to solve the original problem

Running time: by a ***recurrence relation*** combining the size and number of the subproblems and the cost of dividing the problem into the subproblems

19

"Telescoping" a Recurrence

- Recurrence relation and its base condition (i.e., the difference equation and initial condition):

$$\mathrm{T}(n) = 2\cdot\mathrm{T}(n-1) + 1;\ \mathrm{T}(0) = 0$$

- Closed (explicit) form for $\mathrm{T}(n)$ by "telescoping":

20

We substitute the unknown term T(n-1), T(n-2),,, until the base case T(1)=1.

$$\begin{aligned}\mathrm{T}(n) &= 2\,\mathrm{T}(n-1)+1\\ \mathrm{T}(n-1) &= 2\,\mathrm{T}(n-2)+1\\ &\dots\\ \mathrm{T}(1) &= 2\,\mathrm{T}(0)+1\end{aligned}$$

$$\begin{aligned}T(n) &= 2(2T(n-2)+1)+1\\ T(n) &= 2^2T(n-2)+2^1+1\\ T(n) &= 2^3T(n-3)+2^2+2^1+1\\ &\dots\\ T(n) &= 2^nT(0)+2^{n-1}+\dots+2+1\end{aligned}$$

$$\longrightarrow\quad T(n) = 2^n - 1$$

21

"Telescoping" ≡ Substitution

$$\begin{aligned}\mathrm{T}(n) &= 2\,\mathrm{T}(n-1)+1\\ 2\,\mathrm{T}(n-1) &= 2^2\,\mathrm{T}(n-2)+2\\ 2^2\,\mathrm{T}(n-2) &= 2^3\,\mathrm{T}(n-3)+2^2\\ &\dots\\ 2^{n-1}\,\mathrm{T}(2) &= 2^n\,\mathrm{T}(0)+2^{n-1}\end{aligned}$$

$$\mathrm{T}(n) = 1+2+2^2+\dots+2^{n-1} = 2^n-1$$

22

Basic Recurrence: 1

$$\mathrm{T}(n) = \mathrm{T}(n-1)+n \iff \mathrm{T}(n) = \frac{n(n+1)}{2}$$

$$\begin{aligned}\mathrm{T}(n) &= \mathrm{T}(n-1)+n\\ \mathrm{T}(n-1) &= \mathrm{T}(n-2)+n-1\\ &\dots\\ \mathrm{T}(2) &= \mathrm{T}(1)+2\\ \mathrm{T}(1) &= 1\end{aligned}$$

23

1: Explicit Expression for $\mathrm{T}(n)$

$$\begin{aligned}\mathrm{T}(n) &= \mathrm{T}(n-1)+n\\ &= \overline{\mathrm{T}(n-2)+(n-1)}+n\\ &= \overline{\mathrm{T}(n-3)+(n-2)}+(n-1)+n\\ \dots &= \overline{\mathrm{T}(2)+3}+\dots+(n-2)+(n-1)+n\\ &= \overline{\mathrm{T}(1)+2}+\dots+(n-2)+(n-1)+n\\ &= \bar{1}+2+\dots+(n-2)+(n-1)+n = \frac{n(n+1)}{2}\end{aligned}$$

24

Guessing to Solve a Recurrence

- Guess a hypothetic solution $T(n)$; $n \geq 0$ from a sequence of numbers $T(0)$, $T(1)$, $T(2)$, …, obtained from the recurrence relation
- Prove $T(n)$ by math induction:

  Base condition: T holds for $n = n_{base}$,e.g. $T(0)$ or $T(1)$

  Induction hypothesis to verify: for every $n > n_{base}$, if T holds for $n - 1$, then T holds for n

  Strong induction: if T holds for n_{base}, …, n - 1, then…

Explicit Expression for $T(n)$

- $T(1) = 1$; $T(2) = 1 + 2 = 3$; $T(3) = 3 + 3 = 6$; $T(4) = 6 + 4 = 10 \Rightarrow$ Hypothesis:
- Base condition holds: $T(1) = 1 \cdot 2 / 2 = 1$
- If the hypothetic closed-form relationship $T(n)$ holds for $n - 1$ then it holds also for n:
- Thus, the expression for $T(n)$ holds for all $n > 1$

Basic Recurrence: 2

- **Repeated halving principle**: halve the input in one step:

$$T(n) = T(n/2) + 1 \quad \Leftrightarrow \quad T(n) \cong \log_2 n$$

- "Telescoping" (for $n = 2^m$):

$$T(2^m) = T(2^{m-1}) + 1 \qquad T(2^2) = T(2^1) + 1$$
$$T(2^{m-1}) = T(2^{m-2}) + 1 \qquad T(2^1) = T(2^0) + 1$$
$$\ldots \qquad T(2^0) = 0$$

2: Explicit Expression for $T(n)$

$$\begin{aligned} T(2^m) &= T(2^{m-1}) + 1 \\ &= \overline{T(2^{m-2}) + 1} + 1 \\ &\ldots = \overline{T(2^1) + 1} + 1 + \ldots + 1 \\ &= \overline{T(2^0) + 1} + 1 + \ldots + 1 + 1 \end{aligned}$$

$$T(2^m) = m \quad \Rightarrow \quad T(n) = \log_2 n$$

Basic Recurrence: 3

- Scan and halve the input:

$$T(n) = T(n/2) + n \quad \Leftrightarrow \quad T(n) \cong 2n$$

- "Telescoping" (for $n = 2^m$):

$$T(2^m) = T(2^{m-1}) + 2^m \qquad T(2^2) = T(2^1) + 2^2$$
$$T(2^{m-1}) = T(2^{m-2}) + 2^{m-1} \qquad T(2^1) = T(2^0) + 2^1$$
$$\ldots \qquad T(2^0) = 1$$

3: Explicit Expression for $T(n)$

$$\begin{aligned} T(2^m) &= T(2^{m-1}) + 2^m \\ &= \overline{T(2^{m-2}) + 2^{m-1}} + 2^m \\ &\ldots = \overline{T(2^1) + 2^2} + \ldots + 2^{m-1} + 2^m \\ &= \overline{T(2^0) + 2^1} + 2^2 + \ldots + 2^{m-1} + 2^m \end{aligned}$$

$$T(2^m) = 2^{m+1} - 1 \quad \Rightarrow \quad T(n) \cong 2n$$

Basic Recurrence: 4

- **"Divide-and-conquer" prototype:**

$$\mathrm{T}(n) = 2\,\mathrm{T}(n/2) + n \quad \Leftrightarrow \quad \mathrm{T}(n) \cong n \log_2 n$$

- "Telescoping": $\dfrac{\mathrm{T}(n)}{n} = \dfrac{\mathrm{T}(n/2)}{n/2} + 1; \quad \mathrm{T}(1) = 0$

- For $n = 2^m \rightarrow \dfrac{\mathrm{T}(2^m)}{2^m} = \dfrac{\mathrm{T}(2^{m-1})}{2^{m-1}} + 1$

4: Explicit Expression for $\mathrm{T}(n)$

$$\begin{aligned}
\mathrm{T}(2^m)/2^m &= \mathrm{T}(2^{m-1})/2^{m-1} + 1 \\
&= \overline{\mathrm{T}(2^{m-2})/2^{m-2} + 1} + 1 \\
&\ldots = \overline{\mathrm{T}(2^1)/2^1 + 1} + \ldots + 1 + 1 \\
&= \overline{\mathrm{T}(2^0)/2^0 + 1} + 1 + \ldots + 1 + 1 \\
&= 0 + 1 + \ldots + 1 = m
\end{aligned}$$

$$\mathrm{T}(2^m) = m \cdot 2^m \quad \Rightarrow \quad \mathrm{T}(n) = n \log_2 n$$

General "Divide-and-Conquer"

Theorem: The recurrence

with integer constants $a \geq 1$ and $b \geq 2$ and positive constants c and k has the solution:

$$\mathrm{T}(n) = \begin{cases} O(n^{\log_b a}) & \text{if} \quad b^k < a \\ O(n^k \log n) & \text{if} \quad b^k = a \\ O(n^k) & \text{if} \quad b^k > a \end{cases}$$

Proof by telescoping:

$$n = b^m \Rightarrow \mathrm{T}(b^m) = a\,\mathrm{T}(b^{m-1}) + cb^{mk}$$

General "Divide-and-Conquer"

- Telescoping:

$$\begin{aligned}
T(b^m) &= aT(b^{m-1}) + cb^{mk} \\
aT(b^{m-1}) &= a^2T(b^{m-2}) + acb^{(m-1)k} \\
a^2T(b^{m-2}) &= a^3T(b^{m-3}) + acb^{(m-2)k} \\
\ldots &= \ldots \\
a^{m-1}T(b) &= a^mT(1) + a^{m-1}cb^k
\end{aligned}$$

$$\begin{aligned}
\mathrm{T}(b^m) &= a^m c + a^{m-1}cb^k + \ldots + acb^{(m-1)k} + cb^{mk} \\
&= c\sum_{t=0}^{m} a^{m-t}b^{tk} = ca^m \sum_{t=0}^{m} \left(b^k/a\right)^t
\end{aligned}$$

Capabilities and Limitations

- Rough complexity analysis cannot result immediately in an efficient practical program but it helps in predicting of empirical running time of the program
- "Big-Oh" analysis is unsuitable for small input and hides the constants c and n_0 crucial for a practical task
- "Big-Oh" analysis is unsuitable if costs of access to input data items vary and if there is lack of sufficient memory
- But complexity analysis provides ideas how to develop new efficient methods

Data Sorting

- **Ordering relation:** places each pair α, β of ***countable*** items in a fixed order denoted as (α,β) or $<\alpha,\beta>$
- **Order notation:** $\alpha \leq \beta$ (*less than or equal to*)
- **Countable item:** labelled by a specific ***integer key***
- **Comparable objects in Java:** if an object can be ***less than***, ***equal to***, or ***greater than*** another object:

object1.compareTo(object2) **<0, =0, >0**

Order of Data Items

- **Numerical order** - by value:
  $5 \le 5 \le 6.45 \le 22.79 \le \ldots \le 1056.32$
- **Alphabetical order** - by position in an alphabet:
  a ≤ b ≤ c ≤ d ≤ … ≤ z
  Such ordering depends on the alphabet used: look into any bilingual dictionary...
- **Lexicographic order** - by first differing element:
  5456 ≤ 5457 ≤ 5500 ≤ 6100 ≤ …
  pork ≤ ward ≤ word ≤ work ≤ …

37

Features of ordering

- Relation on an array $\mathbf{A} = \{a, b, c, \ldots\}$ is:
  - ***reflexive:*** $a \le a$
  - ***transitive:*** if $a \le b$ and $b \le c$, then $a \le c$
  - ***symmetric:*** if $a \le b$ and $b \le a$, then $a = b$
- **Linear order** if for any pair of elements a and b either $a \le b$ or $b \le a$: $a \le b \le c \le \ldots$
- **Partial order** if there are incomparable elements

38

Insertion Sort

- Splits an array into a unordered and ordered parts
- Sequentially contracts the unordered part, one element per stage:

  ordered part: $a_0, \ldots, a_{i-1}$ unordered part: $a_i, \ldots, a_{n-1}$
- Stage $i = 1, \ldots, n-1$:

  $n-i$ unordered and i ordered elements

39

Example of Insertion Sort

- N_c - number of comparisons per insertion
- N_m - number of moves per insertion

13	18	35	44	15	10	20	N_c	N_m
			15	44			<	→
		15	35				<	→
	15	18					<	→
15							≥	
13	15	18	35	44	10	20	4	3

40

Example of Insertion Sort

13	15	18	35	44	10	20	N_c	N_m
				10	44		<	→
			10	35			<	→
		10	18				<	→
	10	15					<	→
10	13						<	→
10	13	15	18	35	44	20	5	5

41

Implementation of Insertion Sort

```
begin InsertionSort ( integer array a of size n )
1.    for i ← 1 while i < n  step i ← i + 1 do
2.        s_tmp ← a[ i ];     k ← i − 1
3.        while k ≥ 0 AND s_tmp < a[k] do
4.            a[ k +1 ] ← a[ k ];      k ← k − 1
5.        end while
6.        a[ k+1 ] ← s_tmp
7.    end for
 end InsertionSort
```

42

Average Complexity at Stage i

- $i+1$ positions to place a next item: 0 1 2 … i -1 i
- $i-j+1$ comparisons and $i-j$ moves for each position $j = i, i-1, \ldots, 1$
- i comparisons and i moves for position $j = 0$
- Average number of comparisons:

$$E_i = \frac{1+2+\ldots+i+i}{i+1} = \frac{i}{2} + \frac{i}{i+1}$$

Total Average Complexity

- $n-1$ stages for n input items: the total average number of comparisons:

$$E = E_1 + E_2 + \ldots + E_{n-1} = \frac{n^2}{4} + \frac{3n}{4} - H_n$$
$$\left(\tfrac{1}{2}+\tfrac{1}{2}\right)+\left(\tfrac{2}{2}+\tfrac{2}{3}\right)+\ldots+\left(\tfrac{n-1}{2}+\tfrac{n-1}{n}\right)$$
$$= \tfrac{1}{2}(1+2+\ldots+(n-1)) + \left(\tfrac{1}{2}+\tfrac{2}{3}+\ldots+\tfrac{n-1}{n}\right)$$
$$= \tfrac{(n-1)n}{4} + n - \left(\tfrac{1}{2}+\tfrac{1}{3}+\ldots+\tfrac{1}{n}\right)$$

- $H_n \cong \ln n + 0.577$ when $n \to \infty$ is the n-th **harmonic** number

Analysis of Inversions

- An **inversion** in an array $\mathbf{A} = [a_1, a_2, \ldots, a_n]$ is any ordered pair of positions (i, j) such that $i < j$ but $a_i > a_j$: e.g., […, 2,…, 1] or [100, …, 35, …]

$\mathbf{A}$	# invers.	$\mathbf{A}_{\text{reverse}}$	# invers.	Total #
3,2,5	1	5,2,3	2	3
3,2,5,1	4	1,5,2,3	2	6
1,2,3,4,7	0	7,4,3,2,1	10	10

Analysis of Inversions

- Total number of inversions both in an arbitrary array $\mathbf{A}$ and its reverse $\mathbf{A}_{\text{reverse}}$ is equal to the total number of the ordered pairs ($i < j$):

$$\binom{n}{2} = \frac{(n-1)\cdot n}{2}$$

- A sorted array has no inversions
- A reverse sorted array has $\frac{(n-1)n}{2}$ inversions

Analysis of Inversions

- Exactly one inversion is removed by swapping two neighbours $a_{i-1} > a_i$
- An array with k inversions results in $O(n+k)$ running time of insertionSort
- Worst-case time: $c\frac{n^2}{2}$, or $O(n^2)$
- Average-case time: $c\frac{n^2}{4}$, or $O(n^2)$

More Efficient Shell's Sort

- **Efficient sort** must eliminate more than just one inversion between the neighbours per exchange! (insertion sort eliminates one inversion per exchange)
- D.Shell (1959): compare first the keys at a distance of gap_T, then of $\text{gap}_{T-1} < \text{gap}_T$, and so on until of $\text{gap}_1 = 1$
- After a stage with gap_t, all elements spaced gap_t apart are sorted; it can be proven that any gap_t-sorted array remains gap_t-sorted after being then gap_{t-1}-sorted

Implementation of ShellSort

```
begin  Shell Sort ( integer array a of size n )
1.    for gap ← ⌊n/2⌋ while gap > 0
       step gap ← ( if gap = 2 then 1 else ⌊gap/2.2⌋ ) do
2.      for i ← gap while i < n step i ← i + 1 do
3.        s_tmp ← a[i];  k ← i
4.        while( k ≥ gap AND s_tmp < a[ k – gap ]  do
5.           a[ k ] ← a[ k – gap ];   k ← k – gap
6.        end while
7.        a[ k ] ← s_tmp;  8.  end for; 9.  end for
end Shell Sort
```


Example of ShellSort: step 1

gap	*i*:C:M	Data to be sorted									
		25	8	2	91	70	50	20	31	15	65
5	5:1:0	25					50				
	6:1:0		8					20			
	7:1:0			2					31		
	8:1:1				**15**					**91**	
	9:1:1					**65**					**70**
		25	8	2	15	65	50	20	31	91	70

Example of ShellSort: step 2

gap	*i*:C:M	25	8	2	15	65	50	20	31	91	70
2	2:1:1	**2**		**25**							
	3:1:0		8		15						
	4:1:0			25		65					
	5:1:0				15		50				
	6:3:2	2		**20**		**25**		**65**			
	7:2:1				15		**31**		**50**		
	8:1:0							65		91	
	9:1:0								50		70

Example of ShellSort: step 3

gap	*i*:C:M	2	8	20	15	25	31	65	50	91	70
1	1:1:0	2	8								
	2:1:0		8	20							
	3:2:1		8	**15**	**20**						
	4:1:0				20	25					
	5:1:0					25	31				
	6:1:0						31	65			
	7:2:1						31	**50**	**65**		
	8:1:0								65	91	
	9:2:1								65	**70**	**91**

Time complexity of ShellSort

- Heavily depends on gap sequences
- Shell's sequence: $^{n}/_{2}, ^{n}/_{4}, \ldots, 1$:
  $O(n^2)$ worst; $O(n^{1.5})$ average
- "Odd gaps only" (if even: $^{gap}/_{2} + 1$):
  $O(n^{1.5})$ worst; $O(n^{1.25})$ average
- Heuristic sequence: $^{gap}/_{2.2}$: better than $O(n^{1.25})$
- A very simple algorithm with an extremely complex analysis