
Algorithm Design Strategies III

Joaquim Madeira

Version 0.4 – October 2024

Overview

- Tasks from last week – Recap + Questions ?
- The Higher-Lower Game – Analysis
- Example – Empirical Analysis
- **Dynamic Programming**
- Example – Fibonacci's Sequence
- Example – Linear Robot
- Example – Computing Binomial Coefficients
- Memoization
- Timing and Profiling Code

COMPUTING POWERS

BRUTE-FORCE VERSIONS

a^b – Brute-Force – Iterative algorithm

- Compute a^b , with $b \geq 0$, using

$$a^b = a \times a \times \dots \times a$$

- Number of multiplications ?

- Formal + Empirical analysis

a^b – Brute-Force – Iterative algorithm

```
def powerIterV1( a, b ) :  
    """ Computing a**b using a loop """  
  
    assert (type( b ) == int) and (b >= 0), "Wrong exponent!"  
  
    assert (a != 0) or (b != 0), "Cannot compute 0**0 !"  
  
    res = 1  
  
    for i in range( 1, b + 1 ) :  
        res *= a  
  
    return res
```

■ Number of multiplications ?

a^b – Brute-Force – Iterative algorithm

n	$2^{**}n$	#Mults
0	1	0
1	2	1
2	4	2
3	8	3
4	16	4
5	32	5
6	64	6
7	128	7
8	256	8
9	512	9
10	1024	10
11	2048	11

a^b – Brute-Force – Recursive alg.

- Compute a^b , with $b \geq 0$, using
$$a^b = a \times a^{b-1}, \text{ with } a^0 = 1$$
- Number of multiplications ?
 - Formal + Empirical analysis
- Any gains ?

a^b – Brute-Force – Recursive alg.

```
def powerRecV1( a, b ) :  
    """ Computing a**b recursively --- Direct algorithm """  
    assert (type( b ) == int) and (b >= 0), "Wrong exponent!"  
    assert (a != 0) or (b != 0), "Cannot compute 0**0 !"   
  
    if b == 0 :  
        return 1  
  
    return a * powerRecV1( a, b - 1 )
```

■ Number of multiplications ?

a^b – Brute-Force – Recursive alg.

n	$2^{**}n$	#Mults
0	1	0
1	2	1
2	4	2
3	8	3
4	16	4
5	32	5
6	64	6
7	128	7
8	256	8
9	512	9
10	1024	10
11	2048	11

COMPUTING POWERS

DIVIDE-AND-CONQUER

a^b – Divide-And-Conquer

- Compute a^b , with $b \geq 0$, using

$$a^b = a^{b \text{ div } 2} \times a^{(b+1) \text{ div } 2}$$

- Base cases ?
- Always use **two recursive calls** !!
- **Number of multiplications ?**
 - Formal + Empirical analysis
- Is it better than the direct algorithm ?

a^b – Divide-And-Conquer

```
def powerRecV3( a, b ) :  
    """ Computing a**b recursively --- Blind Div & Cong strategy"""  
    # TWO base cases are needed !!  
    # Otherwise, we would not stop when b == 1 !!  
    assert (type( b ) == int) and (b >= 0), "Wrong exponent!"  
    if b == 0 :  
        return 1  
    if b == 1 :  
        return a  
    return powerRecV3( a, b // 2 ) * powerRecV3( a, (b + 1) // 2 )
```

■ Number of multiplications ?

Formal analysis

$$M(n) = M(n \operatorname{div} 2) + M((n+1) \operatorname{div} 2) + 1$$

- Easier to solve if n is a power of 2

- $n = 2^k$, $k = \log_2 n$

$$\begin{aligned} M(n) &= M(n / 2) + M(n / 2) + 1 \\ &= 2 M(n / 2) + 1 = \dots \end{aligned}$$

- Closed formula ? Complexity order ?

a^b – Divide-And-Conquer

n	$2^{**}n$	#Mults
0	1	0
1	2	0
2	4	1
3	8	2
4	16	3
5	32	4
6	64	5
7	128	6
8	256	7
9	512	8
10	1024	9
11	2048	10

COMPUTING POWERS DECREASE-AND-CONQUER

a^b – Decrease-And-Conquer

- Compute a^b , with $b \geq 0$, using

$$a^b = a^{b \div 2} \times a^{b \div 2}, \text{ if } b \text{ is even}$$

$$a^b = a \times a^{(b-1) \div 2} \times a^{(b-1) \div 2}, \text{ if } b \text{ is odd}$$

- Base cases ?
- Use just **ONE recursive call !!**
- **Number of multiplications ?**
 - Formal + Empirical analysis

a^b – Decrease-And-Conquer

```
def powerRecV6( a, b ) :  
    """ Computing a**b recursively --- Smart Dec & Conq strategy """  
    assert (type( b ) == int) and (b >= 0), "Wrong exponent!"  
    if b == 0 :  
        return 1  
    p = powerRecV6( a, b // 2 )  
    if (b % 2) == 0 :  
        return p * p  
    return a * p * p
```

■ Number of multiplications ?

Formal analysis

$$M(n) = M(n \text{ div } 2) + 1, \text{ if } n \text{ is even}$$

$$M(n) = M((n-1) \text{ div } 2) + 2, \text{ if } n \text{ is odd}$$

- Check some examples with pencil and paper
 - Do you understand what is happening?
 - **Best** vs. **worst** cases ?
- Closed formula ? Complexity order ?
- Is it better than the previous algorithms ?

a^b – Decrease-And-Conquer

n	$2^{**}n$	#Mults

0	1	0
1	2	2
2	4	3
3	8	4
4	16	4
5	32	5
6	64	5
7	128	6
8	256	5
9	512	6
10	1024	6
11	2048	7
12	4096	6
13	8192	7
14	16384	7
15	32768	8
16	65536	6
17	131072	7

ARRAY: COUNTING EVEN-VALUED ELEMENTS

Task – Counting – Recap

- Given an **array** with non-negative integer values
- Count the number of **even-valued elements**
- Implement the **3 strategies**:
 - Brute-Force / Dec & C / Div & C
- Formal + Empirical analysis : **Comparisons**

Number of array comparisons ?

■ Brute-Force

- 1 loop / n iterations / n comparisons

■ Decrease & Conquer

- $C(0) = 0$
- $C(n) = 1 + C(n - 1)$

■ Divide & Conquer

- $C(0) = 0$ and $C(1) = 1$
- $C(n) = C(n \text{ div } 2) + C((n+1) \text{ div } 2)$

Number of array comparisons

#Elements	#COMPS	#Even-Values
1	1	0
2	2	1
4	4	2
8	8	4
16	16	8
32	32	16
64	64	32
128	128	64
256	256	128
512	512	256

ARRAY: SEQUENTIAL SEARCH

Task – Sequential Search – Recap

- Given an **array** with non-negative integer values
- Use the **iterative Sequential Search** algorithm to look for a given value
- Formal + Empirical analysis : **Comparisons**
- **Best / Worst / Average Cases ?**

Possible cases ?

■ Best case ?

- 1 comparison $\rightarrow B(n) = O(1)$

■ Worst Case ?

- n comparisons $\rightarrow W(n) = O(n)$

■ Average Case ?

- Various possible scenarios
- BUT always $\rightarrow A(n) = O(n)$

Average Case

- One possible scenario
 - No repeated array values
 - Searched value belongs to the array
 - Can be any array element – Equal probability !
- Average number of comparisons ?
 - How to compute ?
- $A(n) = (n + 1) / 2 \approx n / 2$

A different scenario

Sequential Search on RANDOM arrays of positive integers

Searching RANDOM values

#Elements	#Searches	#Found	#COMPS	#Average_COMPS
1	1	0	1	1.000
2	2	0	4	2.000
4	4	0	16	4.000
8	8	0	64	8.000
16	16	0	256	16.000
32	32	2	993	31.031
64	64	3	4007	62.609
128	128	9	15726	122.859
256	256	56	57621	225.082
512	512	209	207084	404.461
1024	1024	677	658902	643.459
2048	2048	1751	1845775	901.257
4096	4096	4004	3992714	974.784

THE HIGHER-LOWER GAME

The Higher-Lower Game

- Person **A** chooses a **random** integer in $[1, 100]$
- Person **B** **guesses** a number in $[1, 100]$
- Person **A** says: **Yes** / **Low guess** / **High guess**
- Person **B** **keeps guessing** until the answer is Yes
- How long does it take ?
- Let's play the game !!

The Higher-Lower Game

- Naïve strategy vs. **smart** strategy
- How many guesses ?
 - **Best** case ?
 - **Worst** cases ?
 - **Average** cases ?
- Complexity order ?

Task – Game simulation

- **Implement** an iterative function to simulate the game
- **Repeat** the game many times !!
- **Count** the number of **guesses** needed in each game
 - Histogram + Simple statistical analysis
- **Complexity order ?**

One simulation scenario

Simulation: playing the higher-lower game 100000 times
The interval of values is [1, 100]

1 attempts:	975	- 0.97%
2 attempts:	2062	- 2.06%
3 attempts:	3929	- 3.93%
4 attempts:	7935	- 7.94%
5 attempts:	16063	- 16.06%
6 attempts:	31973	- 31.97%
7 attempts:	37063	- 37.06%

MIN	- The smallest number of attempts = 1
MEDIAN	- The "middle" number of attempts = 6
MEAN	- The average number of attempts = 5.8022
MAX	- The largest number of attempts = 7

Another simulation scenario

Simulation: playing the higher-lower game 100000 times
The interval of values is [1, 1000]

1 attempts:	95	- 0.10%
2 attempts:	185	- 0.18%
3 attempts:	379	- 0.38%
4 attempts:	826	- 0.83%
5 attempts:	1612	- 1.61%
6 attempts:	3155	- 3.16%
7 attempts:	6403	- 6.40%
8 attempts:	12819	- 12.82%
9 attempts:	25820	- 25.82%
10 attempts:	48706	- 48.71%

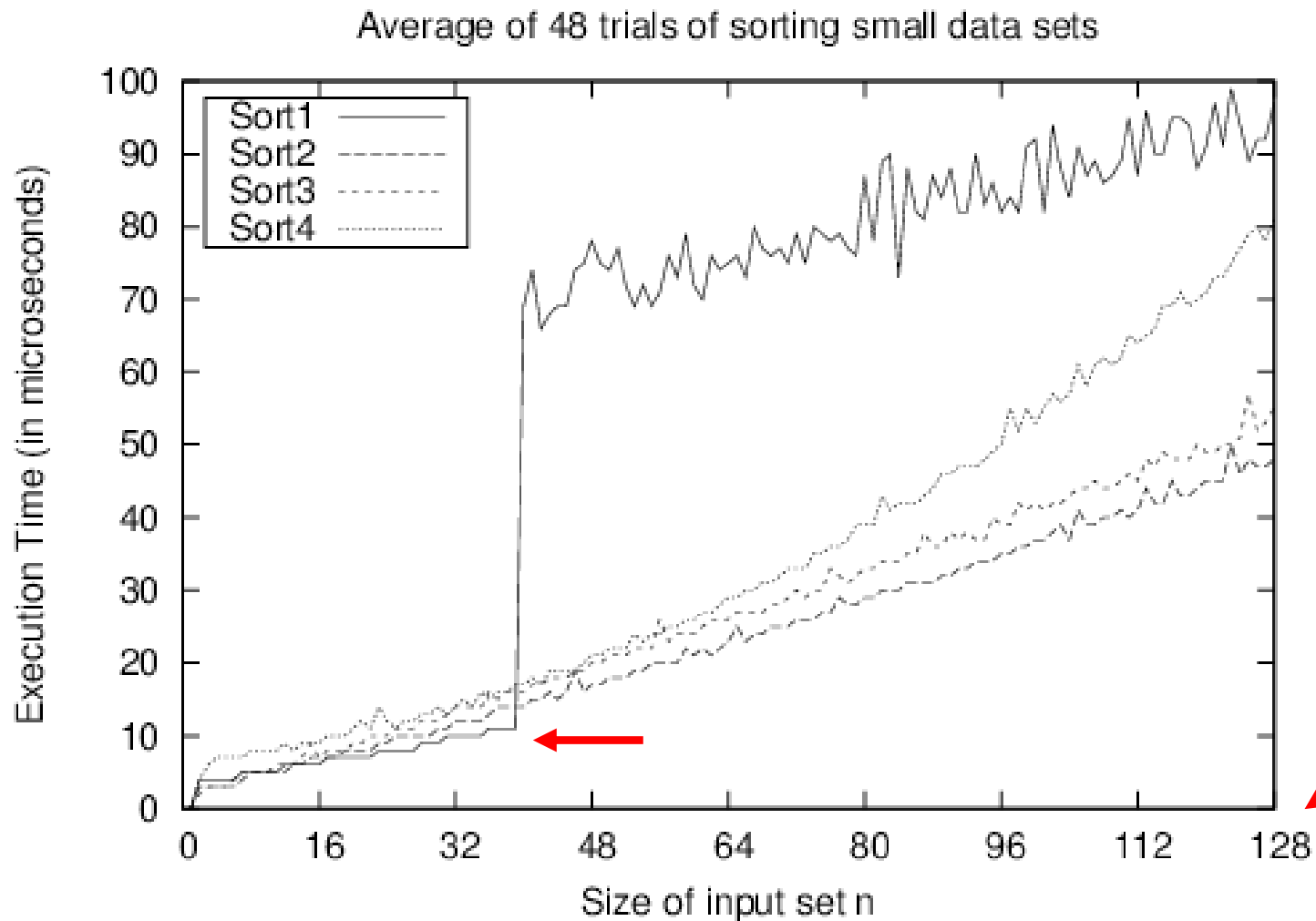
MIN	- The smallest number of attempts = 1
MEDIAN	- The "middle" number of attempts = 9
MEAN	- The average number of attempts = 8.98709
MAX	- The largest number of attempts = 10

EXPERIMENTAL ANALYSIS - AN EXAMPLE

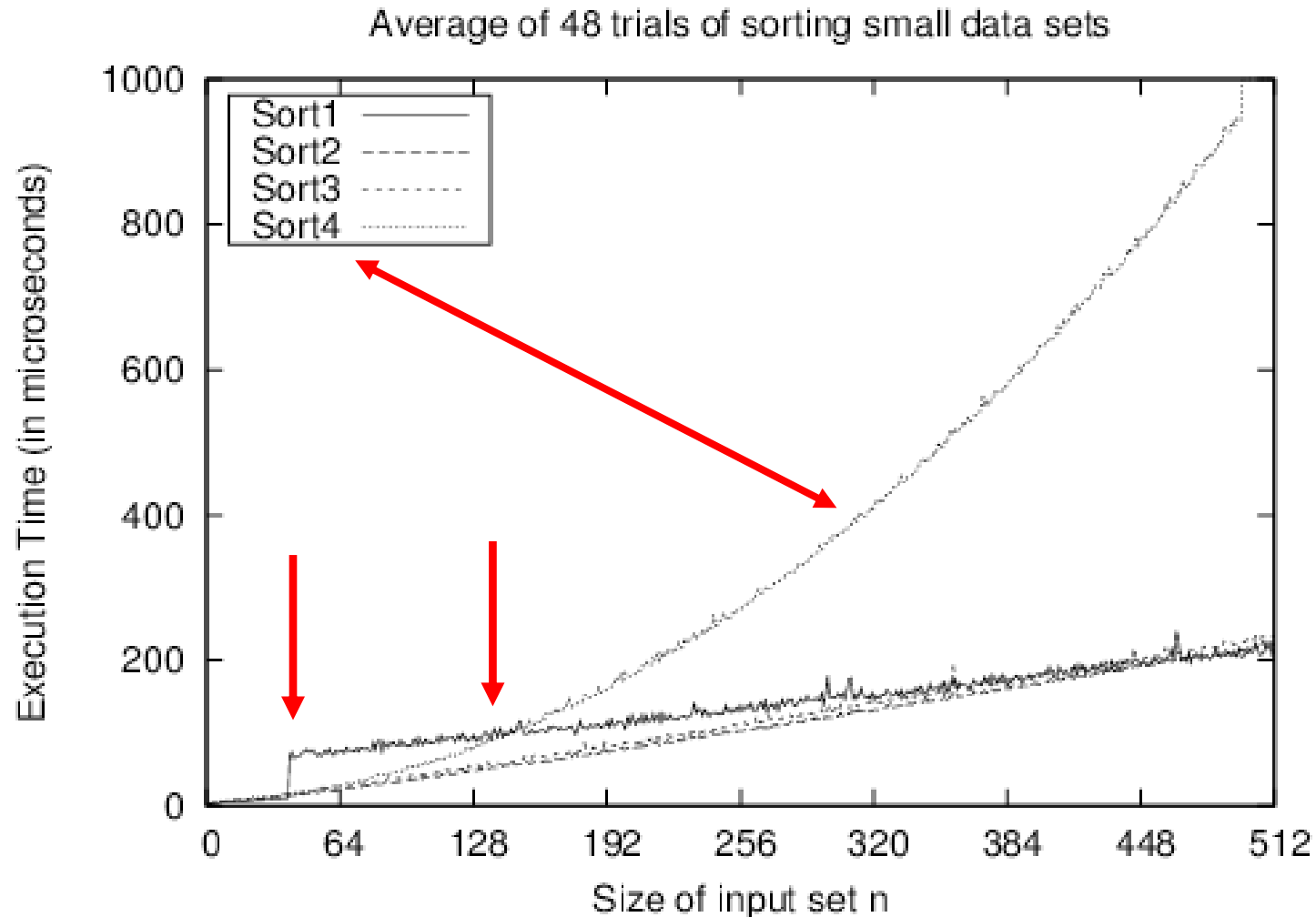
Example – Experimental Analysis

- Performance data for 4 sorting algorithms
- Sorting sets of n random strings
- 50 trials
- Best time and worst time were discarded
- Best algorithm ?
- Complexity order ?
- Identify the algorithms ?
- Heineman et al. Algorithms in a Nutshell. 2nd Ed., O'Reilly, 2016

Average running time – Random data



Average running time – Random data



Regression analysis

- Sort-4

- $y = 0.0053 \times n^2 - 0.3601 \times n + 39.212$

- Sort-2

- $y = 0.05765 \times n \times \log(n) + 7.9653$

- Sort-3

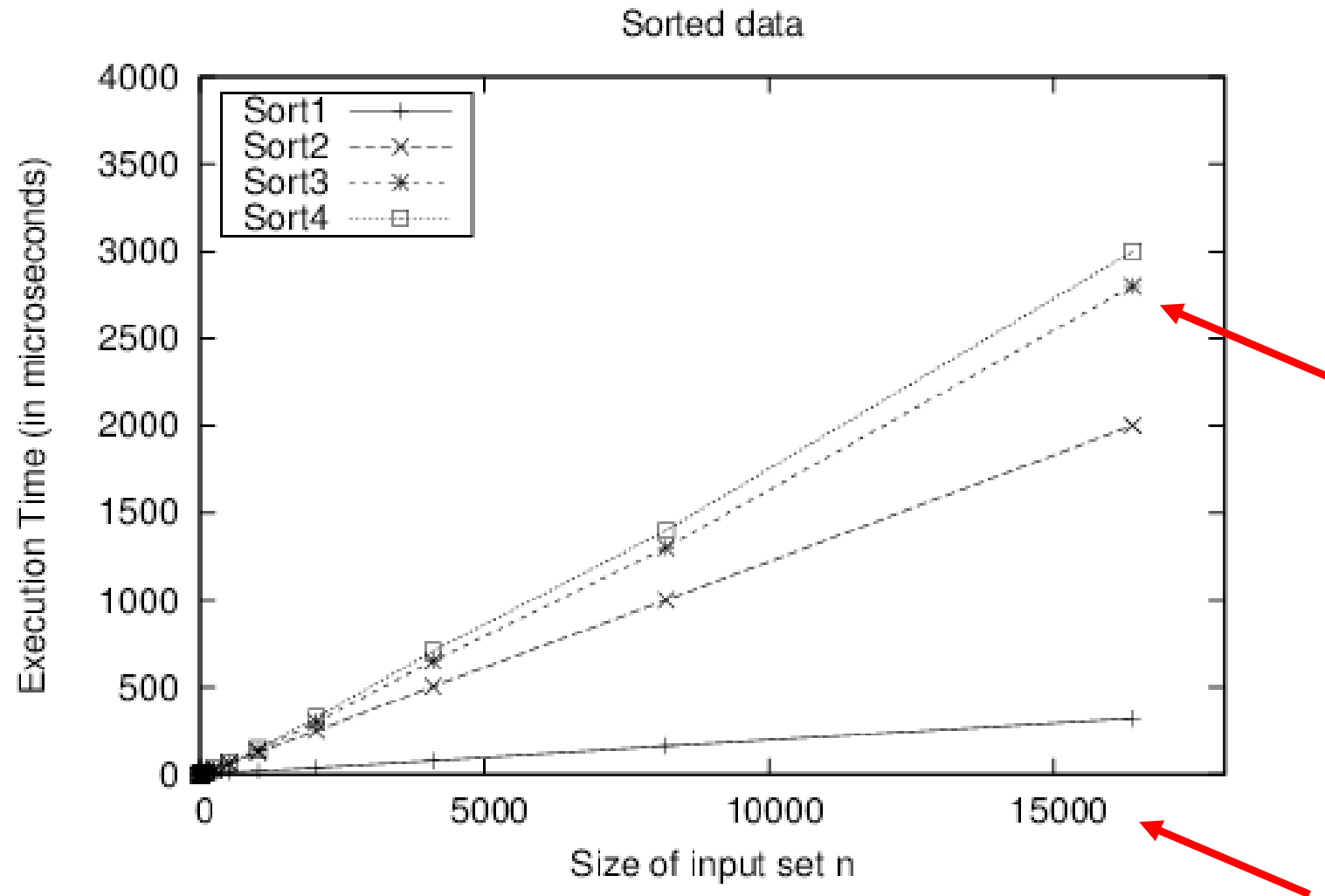
- Slightly slower than Sort-2

- Sort-1

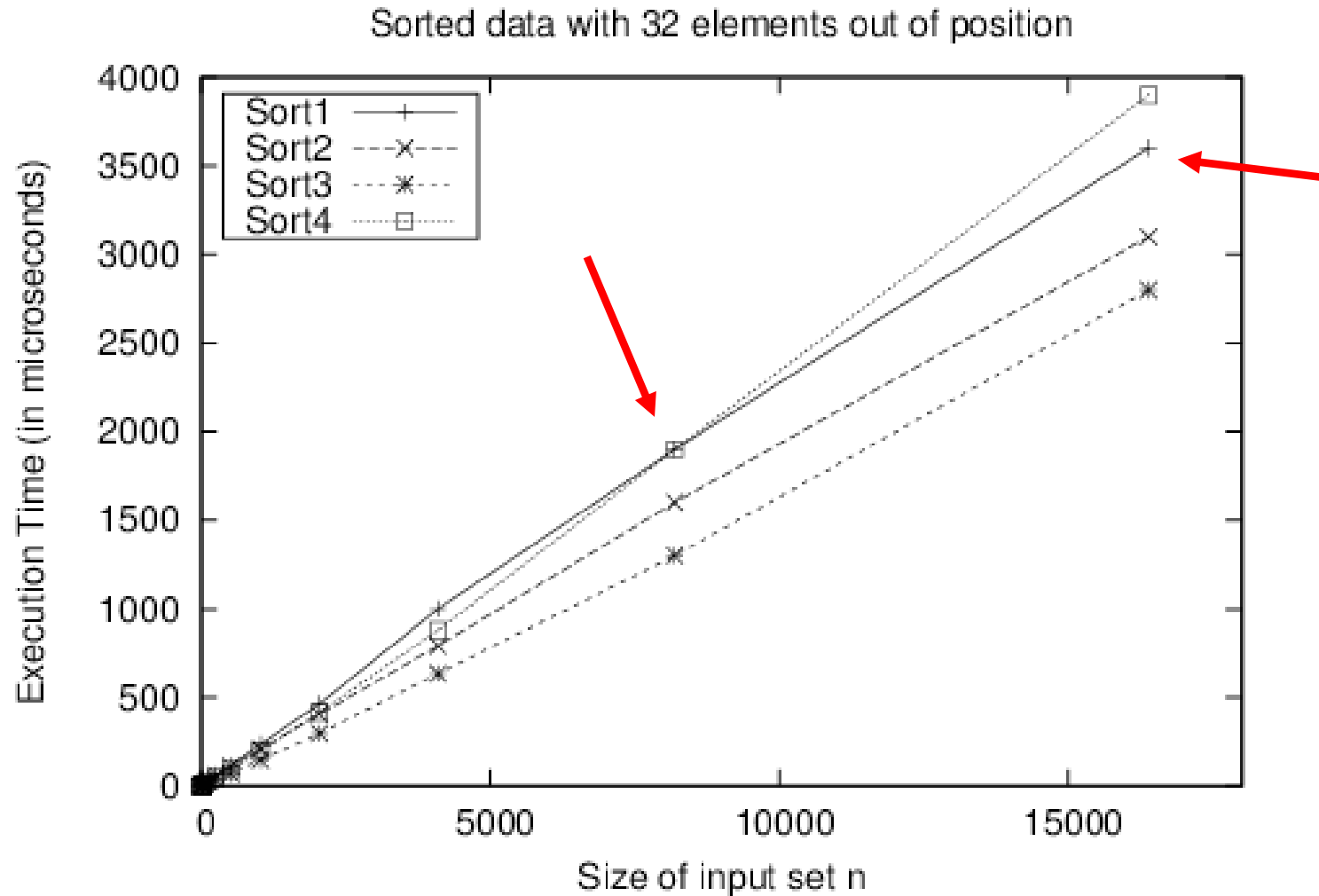
- $n < 40 : y = 0.0016 \times n^2 + 0.2939 \times n + 3.1838$

- $n \geq 40 : y = 0.0798 \times n \times \log(n) + 142.7818$

Average running time – Sorted data

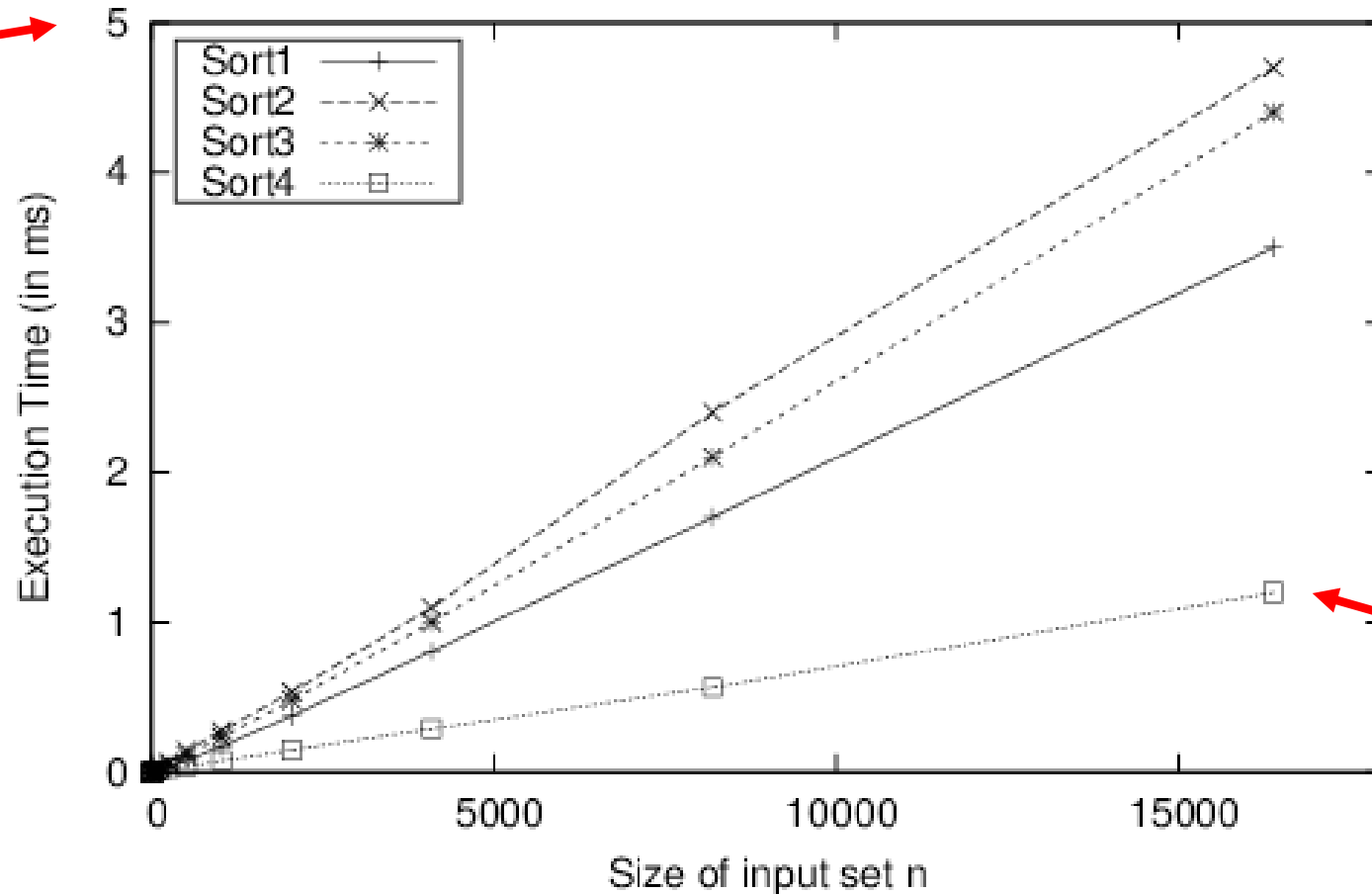


Only 32 elements out of position !



Nearly-sorted

Nearly sorted data where $n/4$ entries are randomly shifted to be 4 away from their proper position



Which algorithms ?

- Sort-1 : **qsort** on Linux
- Sort-2 : ?
- Sort-3 : ?
- Sort-4 : ?

DYNAMIC PROGRAMMING

Dynamic Programming

- General algorithm design technique
- Apply to
 - Computing recurrences
 - Solving optimization problems
- How to **store “previous” results** ?
 - 2D array
 - Vector
 - A few variables

Recurrences – Top-Down

- Exploit the relationship between
 - A solution to a given problem instance
 - Solutions to **smaller/simpler instances** of the same problem
- Set up a **recurrence** !
- Decompose into smaller / simpler **sub-problems**
 - Parameters ?
- Identify the smallest / simplest / **trivial problems**
 - Base cases

Dynamic Programming – Bottom-up

- Use a recurrence: BUT go **bottom-up** !
- **Start** from the smallest / simplest / trivial problems
- Get **intermediate solutions** from smaller / simpler sub-problems
- Which values / results are computed in each step ?
 - How to store ?

Dynamic Programming – Advantage

- Do **sub-problems overlap** ?
- NOW, there is **no need to repeatedly solve** the same sub-problems !!
- Proceed **bottom-up** and **store results** for later use
- Compare with Divide-and-Conquer !!

COMPUTING FIBONACCI NUMBERS

Fibonacci's Sequence

- $F(0) = 0$; $F(1) = 1$
- $F(i) = F(i - 1) + F(i - 2)$; $i = 2, 3, 4, \dots$
- $F(6) = ?$ → Number of recursive calls ?
- Do sub-problems **overlap** ?
- Recursion tree vs. recursion DAG !!
- **Complexity order ?**

Tasks – V1

- **Implement** the recursive function of the previous slide in **Python**
- **Count** the number of **additions** carried out for computing a Fibonacci number
 - Use a **global variable**
- **Table ?**
- **Complexity order ?**

Fibonacci's Sequence

```
def fibonacci_DC( n ) :  
  
    """ Recursive computation of Fi """  
  
    # Global variable, for counting the number of additions  
  
    global num_adds  
  
    if ( n == 0 ) or ( n == 1 ) :  
  
        return n  
  
    num_adds += 1  
  
    return fibonacci_DC( n - 1 ) + fibonacci_DC( n - 2 )
```

Number of additions ?

- $A(0) = 0$; $A(1) = 0$
- $A(i) = 1 + A(i - 1) + A(i - 2)$; $i = 2, 3, 4, \dots$
- Closed formula ?
- You can get it, if you remember Discrete Mathematics...
- BUT, we can get the complexity order from the table...

Additions – Recursive version

- How fast does $F(n)$ grow ?

- How fast does $A(n)$ grow ?

- From the table we get:

$$A(n) = F(n+1) - 1$$

- **Exponential** growth !!

- Why ?

$$(1 + \sqrt{5}) / 2 = 1,618034$$

n	F(n)	Ratio	A(n)	Ratio
0	0		0	
1	1		0	
2	1	1	1	
3	2	2	2	2
4	3	1,5	4	2
5	5	1,666667	7	1,75
6	8	1,6	12	1,714286
7	13	1,625	20	1,666667
8	21	1,615385	33	1,65
9	34	1,619048	54	1,636364
10	55	1,617647	88	1,62963
11	89	1,618182	143	1,625
12	144	1,617978	232	1,622378
13	233	1,618056	376	1,62069
14	377	1,618026	609	1,619681
15	610	1,618037	986	1,619048
16	987	1,618033	1596	1,618661
17	1597	1,618034	2583	1,618421
18	2584	1,618034	4180	1,618273
19	4181	1,618034	6764	1,618182
20	6765	1,618034	10945	1,618125

Fibonacci's Sequence

- $F(0) = 0 ; F(1) = 1$
- $F(i) = F(i - 1) + F(i - 2) ; i = 2, 3, 4, \dots$
- Use Dynamic Programming !!
- Computing $F(n)$ using an **array**
 - Complexity order ?
- Can we use **less** memory space ?

Tasks – V2 + V3

- **Implement** two iterative functions for computing $F(i)$
 - **V2** : using an **array**
 - **V3** : using just **3 variables**
- **Count** the number of **additions** carried out
- **Table ?**
- **Complexity order ?**

Fibonacci's Sequence

i	f(i)	#ADDs-Rec	#ADDs_DP_1	#ADDs_DP_2
0	0	0	0	0
1	1	0	0	0
2	1	1	1	1
3	2	2	2	2
4	3	4	3	3
5	5	7	4	4
6	8	12	5	5
7	13	20	6	6
8	21	33	7	7
9	34	54	8	8
10	55	88	9	9
11	89	143	10	10
12	144	232	11	11
13	233	376	12	12
14	377	609	13	13
15	610	986	14	14

EXAMPLE

- A LINEAR ROBOT

Another example

- Linear robot
- Can move forward by 1 meter, or 2 meters, or 3 meters
- In how many ways can it move a distance of n meters ?
- Establish the recurrence !!
 - Base cases ?

Tasks – $V1 + V2 + V3$


- **Implement** three functions for computing $R(i)$
 - **V1** : using recursion
 - **V2** : using an **array**
 - **V3** : using a few **variables** – how many ?
- **Count** the number of **additions** carried out
 - Formulas ?
- **Tables ?**
- **Complexity order ?**

Example – Results table

i	r(i)	#ADDs-Rec	#ADDs_DP_1	#ADDs_DP_2
1	1	0	0	0
2	2	0	0	0
3	4	0	0	0
4	7	2	2	2
5	13	4	4	4
6	24	8	6	6
7	44	16	8	8
8	81	30	10	10
9	149	56	12	12
10	274	104	14	14
11	504	192	16	16
12	927	354	18	18
13	1705	652	20	20
14	3136	1200	22	22
15	5768	2208	24	24

COMPUTING BINOMIAL COEFFICIENTS

Computing Binomial Coefficients

- $C(n,0) = 1$; $C(n,n) = 1$
- $C(n,j) = C(n-1,j) + C(n-1,j-1)$; $j = 1, 2, \dots, n-1$
- Two arguments !!
- $C(4,3) = ?$  Number of recursive calls ?
- Do sub-problems **overlap** ?
- Recursion tree vs. recursion DAG !!
- **Complexity order ?**

Computing Binomial Coefficients

- **V1** : Compute $C(n,j)$ recursively
- **V2** : Compute $C(n,j)$ using a 2D array
 - How to proceed ?
 - Have you seen this “triangle” before ?
- Can we use **less** memory space ?
- And other, more efficient recurrences ?

Tasks – $V1 + V2 + V3$

- **Implement** three functions for computing $C(n,j)$
 - **V1** : using recursion
 - **V2** : using a 2D **array**
 - **V3** : using a 1D **array**
- **Count** the number of **additions** carried out
- **Tables ?**
- **Complexity order ?**

Pascal's Triangle

Pascal's Triangle - Recursive Function

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10 5  1
1  6 15 20 15 6  1
1  7 21 35 35 21 7  1
1  8 28 56 70 56 28 8  1
1  9 36 84 126 126 84 36 9  1
1 10 45 120 210 252 210 120 45 10  1
```

V1 – Number of additions

Number of Additions - Recursive Function

0											
0	0										
0	1	0									
0	2	2	0								
0	3	5	3	0							
0	4	9	9	4	0						
0	5	14	19	14	5	0					
0	6	20	34	34	20	6	0				
0	7	27	55	69	55	27	7	0			
0	8	35	83	125	125	83	35	8	0		
0	9	44	119	209	251	209	119	44	9	0	

V2 – Number of additions

Number of Additions - Dynamic Programming - V. 1

0											
0	0										
1	1	1									
3	3	3	3								
6	6	6	6	6							
10	10	10	10	10	10						
15	15	15	15	15	15	15					
21	21	21	21	21	21	21	21				
28	28	28	28	28	28	28	28	28			
36	36	36	36	36	36	36	36	36	36		
45	45	45	45	45	45	45	45	45	45	45	45

V3 – Number of additions

Number of Additions - Dynamic Programming - V. 2

0										
0	0									
0	1	0								
0	3	3	0							
0	6	6	6	0						
0	10	10	10	10	0					
0	15	15	15	15	15	0				
0	21	21	21	21	21	21	0			
0	28	28	28	28	28	28	28	0		
0	36	36	36	36	36	36	36	36	0	
0	45	45	45	45	45	45	45	45	45	0

MEMOIZATION

Memoization

- Turning the results of a function into something **to be remembered**
- I.e., **avoid repeating** the calculation of results for previously processed inputs
- Use a table / array / **cache** to store previously computed results
 - Initialization !
- Time vs. space trade-off

Memoization

- Initialize all table entries to “null”
 - Not yet computed
- Whenever a result is to be computed for a given input
 - Check the corresponding table entry
 - If not “null”, retrieve the result
 - Otherwise, compute by a recursive call(s)
 - And store the result

Fibonacci's Sequence

- Initialization

```
for(i=1, i<n, i++) f[i] = -1;
```

- Recursive function

```
int fib( int n ) {  
    int r;  
    if( f[n] != -1 ) return f[n];  
    if( n == 1 ) r = 1;  
    else if( n == 2 ) r = 1;  
    else {  
        r = fib( n - 2 );  
        r = r + fib( n - 1 );  
    }  
    f[n] = r;  
    return r;  
}
```

The “old” Python way

M. Hetland, Python Algorithms, Apress, 2010 - Chapter 8

```
from functools import wraps
```

```
def memo( func ) :
```

```
    cache = {}                                # Stored subproblem solutions
```

```
    @wraps(func)                             # Make wrap look like func
```

```
    def wrap( *args ) :                      # The memoized wrapper
```

```
        if args not in cache :               # Not already computed?
```

```
            cache[args] = func( *args )      # Compute & cache the solution
```

```
        return cache[args]                  # Return the cached solution
```

```
    return wrap                              # Return the wrapper
```

The “old” Python way

i	f(i)	#ADDs_Memo
0	0	0
1	1	0
2	1	1
3	2	1
4	3	1
5	5	1
6	8	1
7	13	1
8	21	1
9	34	1
10	55	1

Testing the memoized version

```
fibonacci_DC = memo( fibonacci_DC )
```

85	259695496911122585	1
86	420196140727489673	1
87	679891637638612258	1
88	1100087778366101931	1
89	1779979416004714189	1
90	2880067194370816120	1

The “new” Python way

- Use the **functools module** and the **cache decorator**

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

TIMING & PROFILING CODE

timeit – Measuring execution time

- Measure execution time of small **code snippets** / **functions**

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

```
>>> import timeit
>>> timeit.timeit("'-'.join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit("'-'.join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit("'-'.join(map(str, range(100)))', number=10000)
0.23702679807320237
```

timeit – Mesuring execution time

- Pass a **setup parameter** to give access to functions you define
 - It contains an import statement

```
def test():  
    """Stupid test function"""  
    L = [i for i in range(100)]  
  
if __name__ == '__main__':  
    import timeit  
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

The Python Profilers

- **cProfile** and **profile** provide deterministic profiling of Python programs
- A profile is a **set of statistics** that describes **how often** and for **how long** various parts of the program executed
- These statistics can be formatted into **reports** via the **pstats** module.

line_profiler – Line-by-line profiling

- Install line_profiler: `pip install line_profiler`
- In the relevant file(s), import line profiler and decorate function(s) you want to profile with `@line_profiler.profile`
- Set the environment variable `LINE_PROFILE=1` and run your script as normal.
- When the script ends a summary of profile results, files written to disk, and instructions for inspecting details will be written to `stdout`

REFERENCES

References

- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd Ed., Pearson, 2012
 - Chapter 8
- R. Johnsonbaugh and M. Schaefer, *Algorithms*, Pearson Prentice Hall, 2004
 - Chapter 8
- T. H. Cormen et al., *Introduction to Algorithms*, 3rd Ed., MIT Press, 2009
 - Chapter 15