# Work

**Definition.**
The work of a program (on a given input) is the sum total of all the operations executed by the program.



# Precomputation

The idea of precomputation is to perform calculations in advance so as to avoid doing them at "mission-critical" times.

Example: Binomial coefficients

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$



Computing the "choose" function by implementing this formula can be expensive (lots of multiplications), and watch out for integer overflow for even modest values of n and k.

Idea: Precompute the table of coefficients when initializing, and perform table look-up at runtime.

# Inlining

The idea of inlining is to avoid the overhead of a function call by replacing a call to the function with the body of the function itself.

```
double square(double x) {
  return x*x;
}

double sum_of_squares(double *A, int n) {
  double sum = 0.0;
  for (int i = 0; i < n; ++i) {
    sum += square(A[i]);
  }
  return sum;
}
```

```
double sum_of_squares(double *A, int n) {
  double sum = 0.0;
  for (int i = 0; i < n; ++i) {
    double temp = A[i];
    sum += temp*temp;
  }
  return sum;
}
```

# Packing and Encoding

The idea of packing is to store more than one data value in a machine word. The related idea of encoding is to convert data values into a representation requiring fewer bits.

## Example: Encoding dates

- The string "September 11, 2018" can be stored in 18 bytes — more than two double (64-bit) words — which must moved whenever a date is manipulated.
- Assuming that we only store years between 4096 B.C.E. and 4096 C.E., there are about $365.25 \times 8192 \approx 3\,\text{M}$ dates, which can be encoded in $\lceil \lg(3 \times 10^6) \rceil = 22$ bits, easily fitting in a single (32-bit) word.
- But determining the month of a date takes more work than with the string representation.

# Why Learn Bit Hacks?

## Why learn bit hacks if they don't even work?

- Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly code.
- Because sometimes the compiler doesn't optimize, and you have to do it yourself by hand.
- Because many bit hacks for words extend naturally to bit and word hacks for vectors.
- Because these tricks arise in other domains, and so it pays to be educated about them.
- Because they're fun!

# Set the kth Bit

## Problem

Set kth bit in a word x to 1.

## Idea

Shift and OR.

```
y = x | (1 << k);
```



## Example

k = 7

| | |
|---|---|
| x | 1011110101101101 |
| 1 << k | 0000000010000000 |
| x \| (1 << k) | 1011110111101101 |

# Clear the kth Bit

## Problem

Clear the kth bit in a word x.

## Idea

Shift, complement, and AND.

```
y = x & ~(1 << k);
```



## Example

k = 7

| | |
|---|---|
| x | 1011110111101101 |
| 1 << k | 0000000010000000 |
| ~(1 << k) | 1111111101111111 |
| x & ~(1 << k) | 1011110101101101 |

# Toggle the kth Bit

## Problem

Flip the k th bit in a word x.

## Idea

Shift and XOR.

```
y = x ^ (1 << k);
```



## Example (0 → 1)

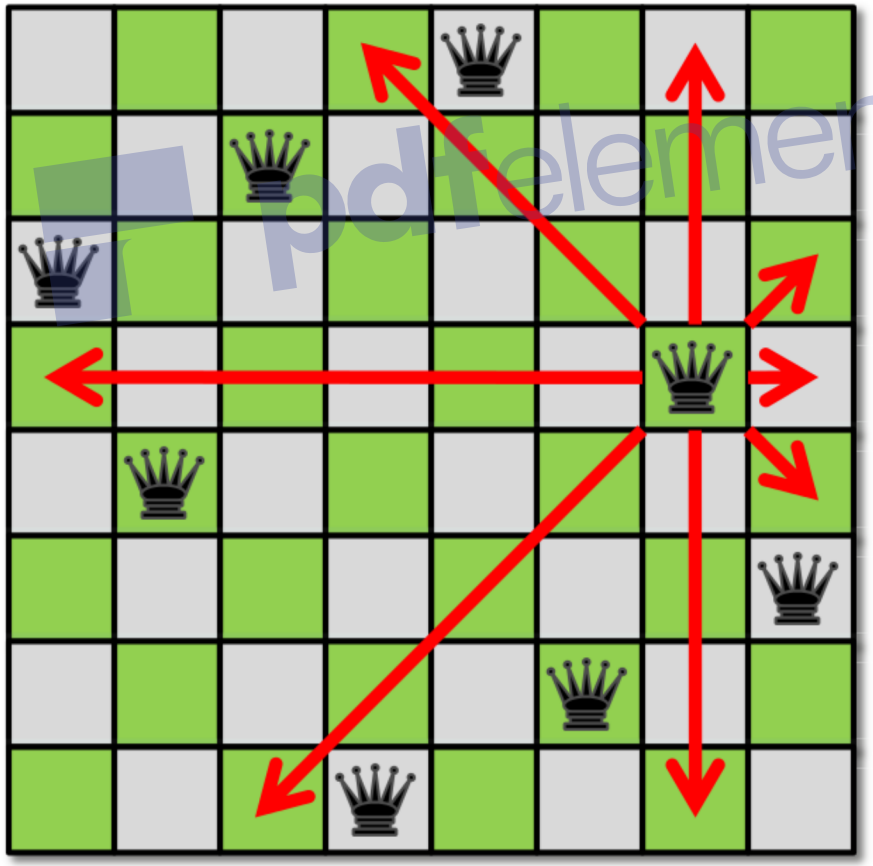
k = 7

| | |
|---|---|
| x | 1011110101101101 |
| 1 << k | 0000000010000000 |
| x ^ (1 << k) | 1011110111101101 |

# Queens Problem

## Problem

Place $n$ queens on an $n \times n$ chessboard so that no queen attacks another, i.e., no two queens in any row, column, or diagonal. Count the number of possible solutions.

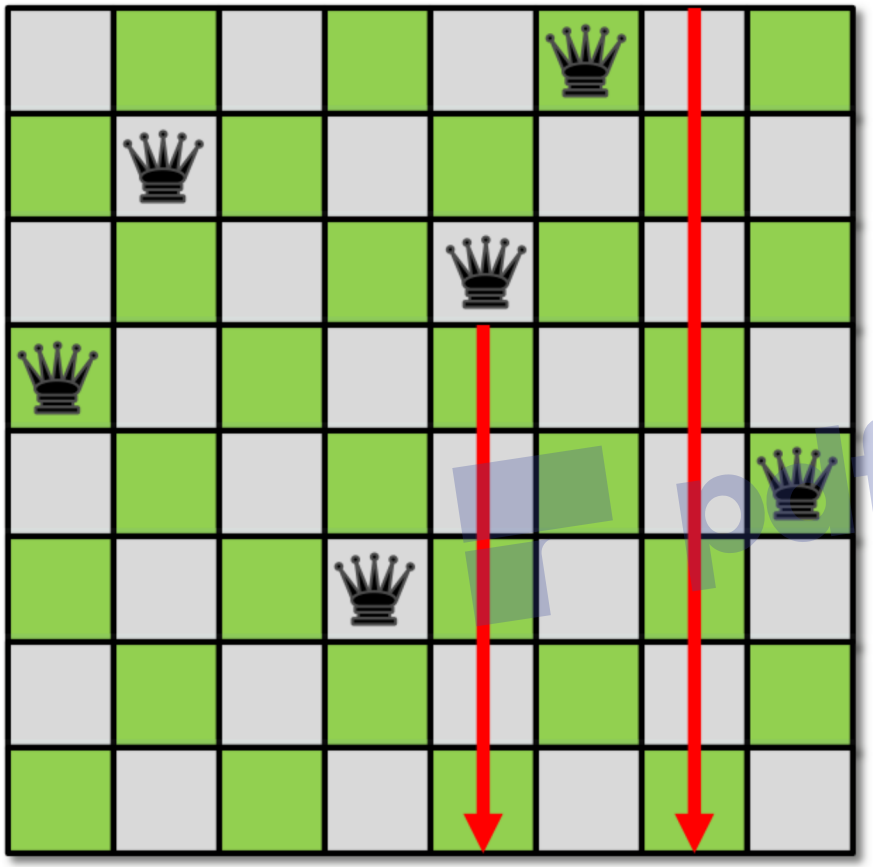


# Board Representation

The backtrack search can be implemented as a simple recursive procedure, but how should the board be represented to facilitate queen placement?

- array of $n^2$ bytes?
- array of $n^2$ bits?
- array of $n$ bytes?
- 3 bitvectors of size $n$, $2n-1$, and $2n-1$.

# Bitvector Representation



1 1 0 1 1 1 0 1

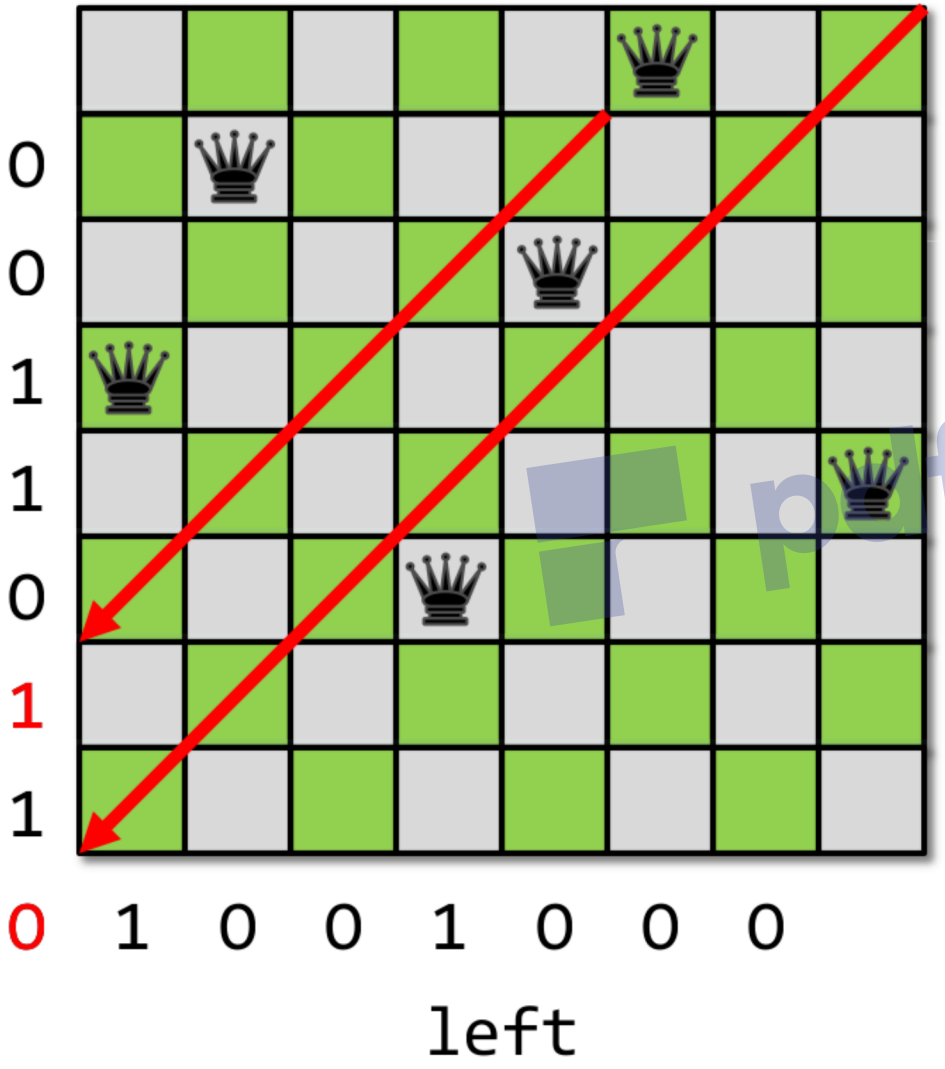
down

Placing a queen in column `c` is not safe if

```
down & (1 << c);
```

is nonzero.

# Bitvector Representation

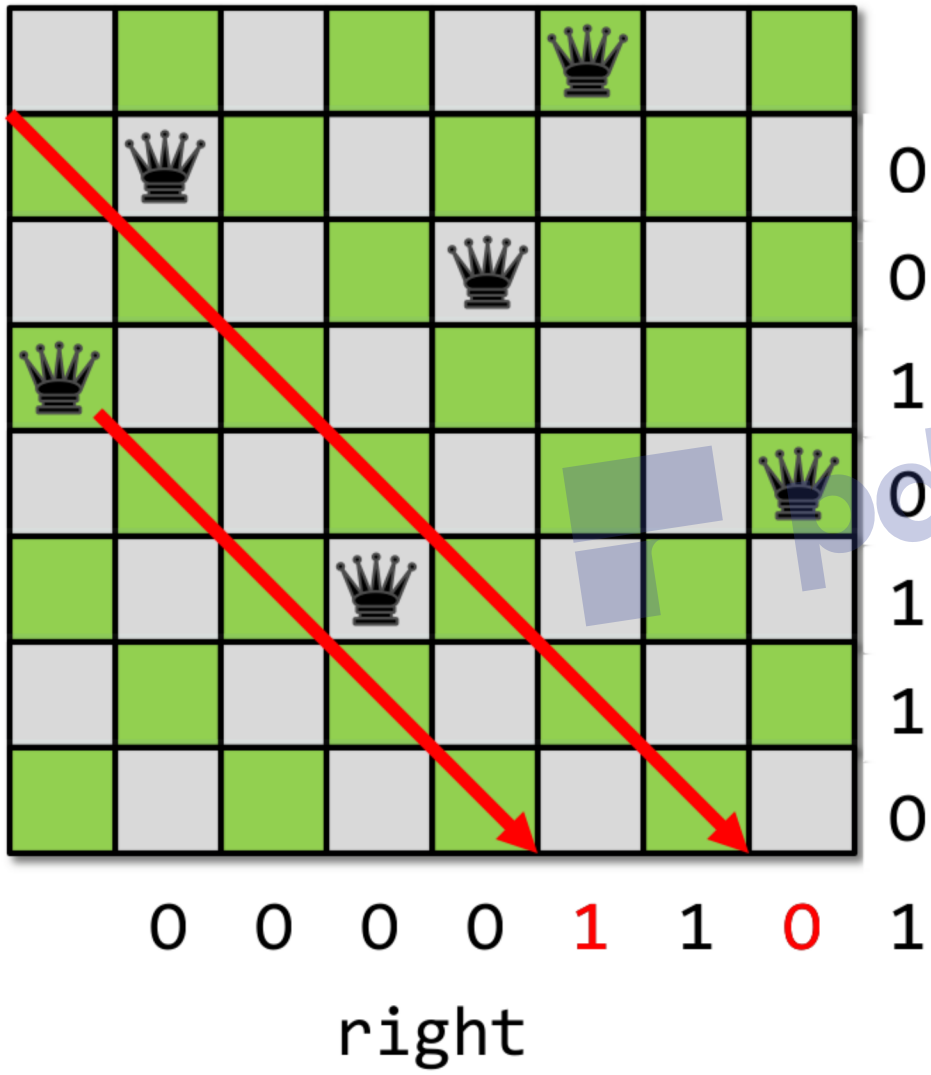



Placing a queen in row `r` and column `c` is not safe if

```
left & (1 << (r+c))
```

is nonzero.

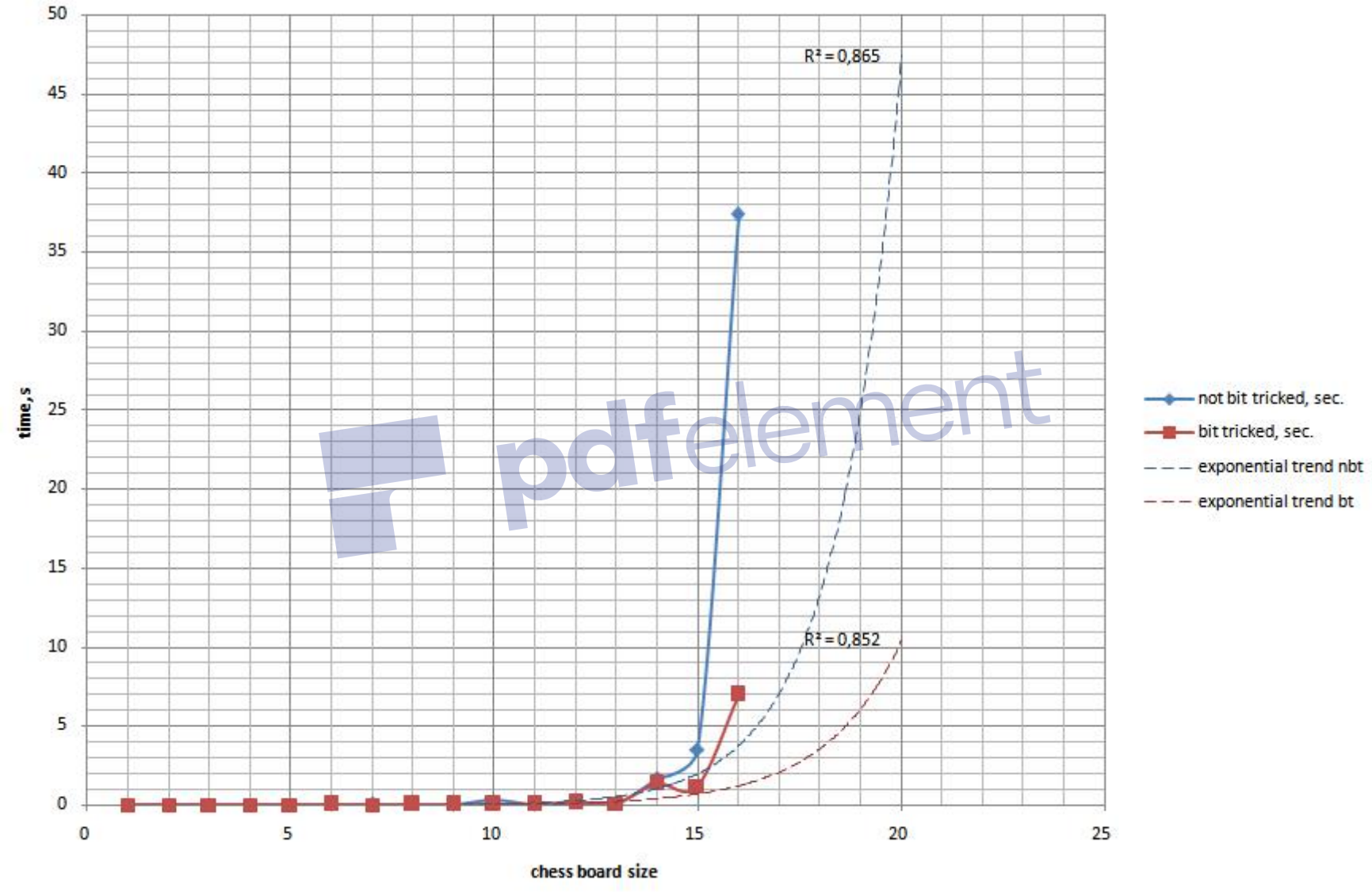
# Bitvector Representation

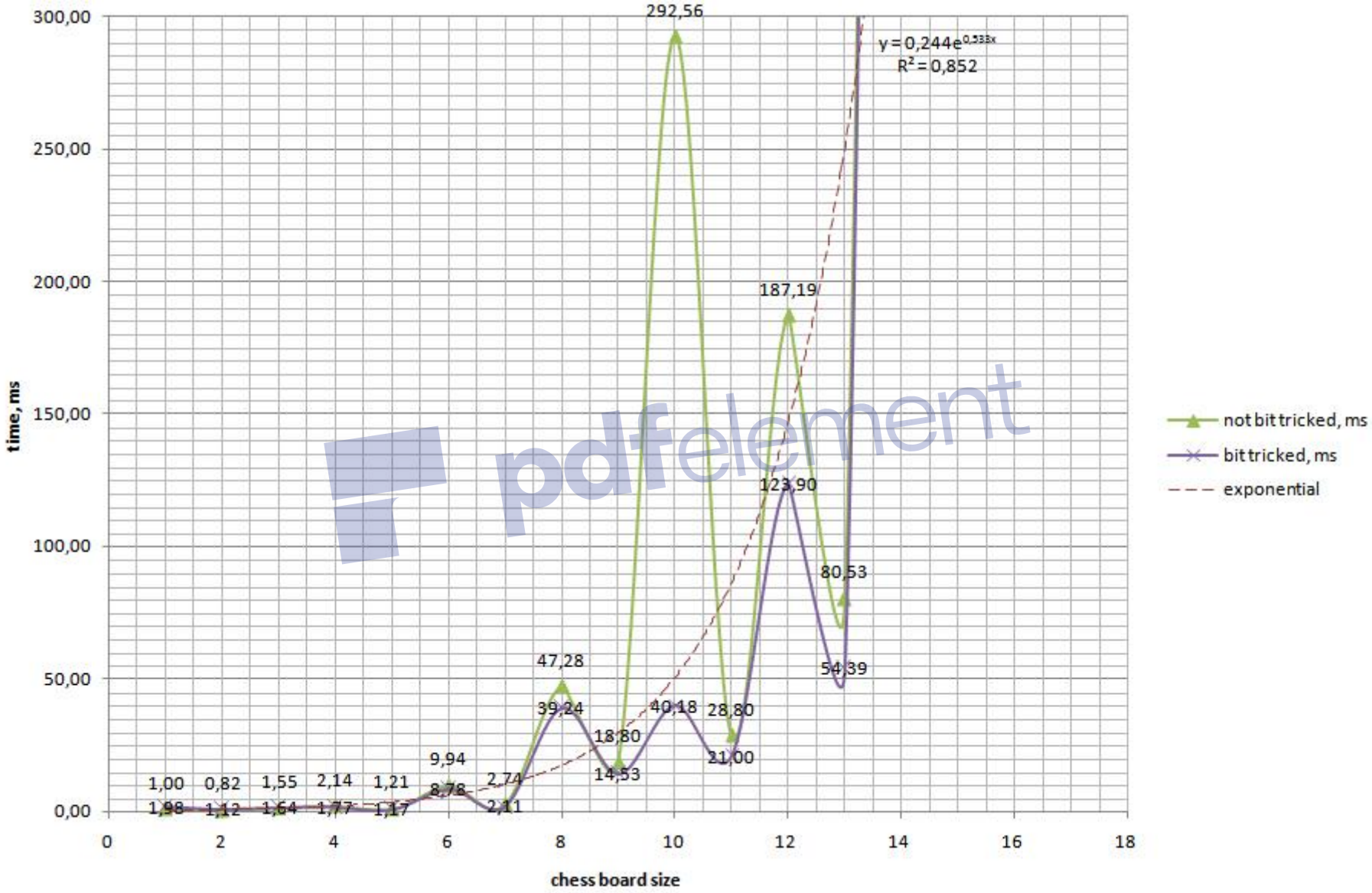



Placing a queen in row `r` and column `c` is not safe if

```
right & (1 << (n-1-r+c))
```

is nonzero.


50
45
40
35
30
25
20
15
10
5
0
time, s
0
5
10
15
20
25
chess board size
R² = 0,865
R² = 0,852
not bit tricked, sec.
bit tricked, sec.
exponential trend nbt
exponential trend bt


300,00
250,00
200,00
150,00
100,00
50,00
0,00
time, ms
0
2
4
6
8
10
12
14
16
18
chess board size
y = 0,244e^0,533x
R² = 0,852
292,56
187,19
123,90
80,53
54,39
47,28
39,24
40,18
28,80
21,00
18,80
14,53
9,94
8,78
2,74
2,11
1,00
0,82
1,55
2,14
1,21
1,98
1,12
1,64
1,77
1,17
not bit tricked, ms
bit tricked, ms
exponential

# Dynamic Frequency and Voltage Scaling

***DVFS*** is a technique to reduce power by adjusting the clock frequency and supply voltage to transistors.

- Reduce operating frequency if chip is too hot or otherwise to conserve (especially battery) power.
- Reduce voltage if frequency is reduced.

$$\text{Power} \propto C\,V^2\,f$$



$C$ = dynamic capacitance
$\approx$ roughly area $\times$ activity (how many bits toggle)
$V$ = supply voltage
$f$ = clock frequency

*Reducing frequency and voltage results in a cubic reduction in power (and heat).*

But it wreaks havoc on performance measurements!

# Sources of Variability

- Daemons and background jobs
- Interrupts
- Code and data alignment
- Thread placement
- Runtime scheduler
- Hyperthreading
- Multitenancy
- Dynamic voltage and frequency scaling (DVFS)
- Turbo Boost
- Network traffic