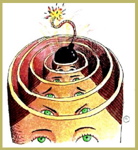
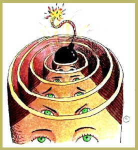


Lecture 25:
Algorithms with Exponential Complexity

Aims:

- To discuss exponential functions;
- To look at a problem whose algorithms have exponential and factorial complexity.



25.1. Exponentiation

- We are used to the idea that multiplication is a process of repeated addition. Similarly, *exponentiation* is a process of repeated multiplication.

  For example,

$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 2 \times 2 = 4$$
$$2^3 = 2 \times 2 \times 2 = 8$$
$$2^4 = 2 \times 2 \times 2 \times 2 = 16$$

  In, for example, 2^4, 2 is the *base* and 4 is the *exponent* and we say "2 *raised to the power* 4 is 16" or "2 raised to the 4th power is 16".

- In general, where c is a positive integer

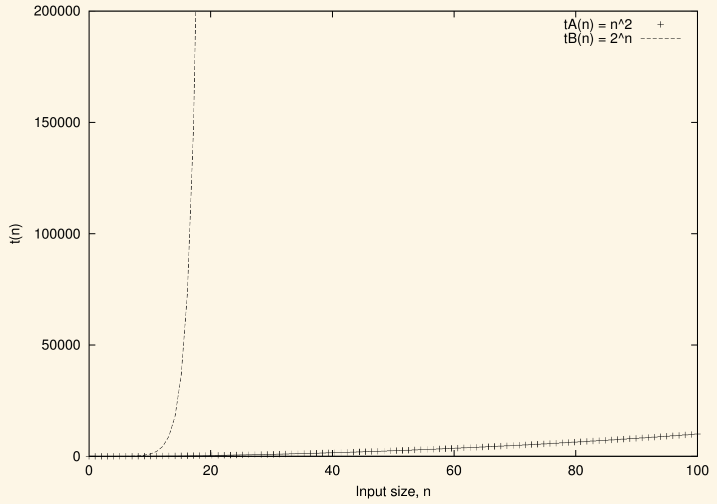
$$\begin{array}{rcl} b^c & =_{\text{def}} & \underbrace{b \times b \times \ldots \times b}_{c \text{ times}} \\ b^0 & =_{\text{def}} & 1 \\ b^{-c} & =_{\text{def}} & 1/b^c \end{array}$$

- Some laws:

$$\begin{array}{rcl} b^{c+d} & = & b^c \times b^d \\ b^{c-d} & = & b^c/b^d \\ b^{cd} & = & (b^c)^d \end{array}$$

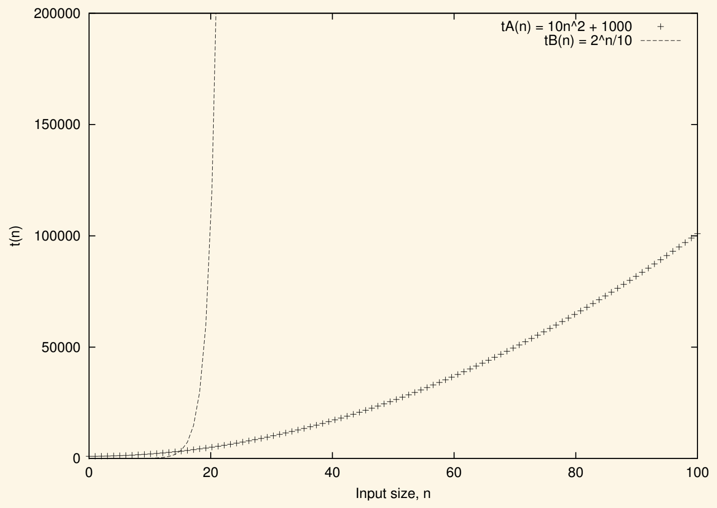
- E.g. simplify $4^n/2^n$

- Suppose algorithm A's worst-case time complexity $t_A(n) =_{\text{def}} n^2$, and algorithm B's worst-case time complexity $t_B(n) =_{\text{def}} 2^n$. 2^n grows much much more quickly than n^2.

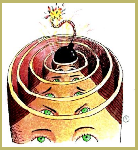


- Multiplying and adding constants and other terms shifts and stretches the graphs. And this may make the curves cross in different places. But the exponential functions will still grow much faster than the polynomial ones.

  For example, suppose algorithm A's worst-case time complexity $t_A(n) =_{\text{def}} 10n^2 + 1000$ and algorithm B's worst-case time complexity $t_B(n) =_{\text{def}} \frac{2^n}{10}$.



For small inputs, algorithm A, whose time complexity is quadratic, takes more time than algorithm B, whose time complexity is exponential. But when input sizes exceed 15, algorithm A becomes the faster and, from that point on, the larger the input, the bigger the advantage A has over B.

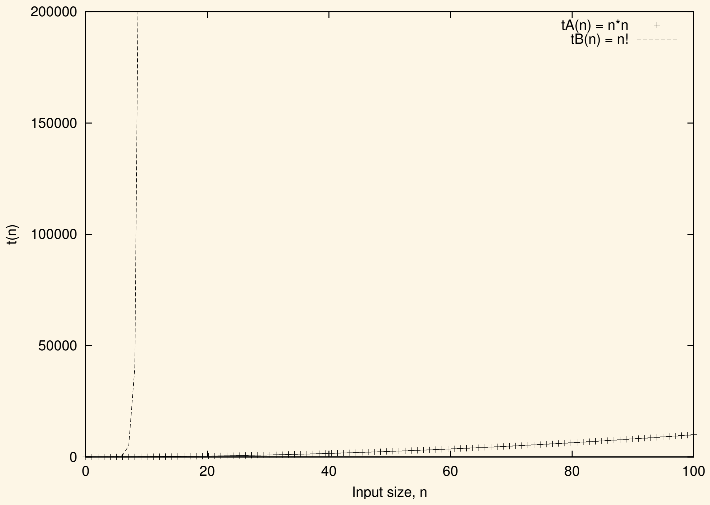


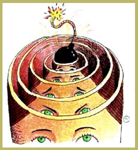
25.2. Factorial

- Factorial is another important function that can crop up when we look at the complexity of algorithms.

$$\begin{aligned} 0! &=_{\text{def}} 1 \\ n! &=_{\text{def}} n \times (n-1) \times \ldots \times 3 \times 2 \times 1 \end{aligned}$$

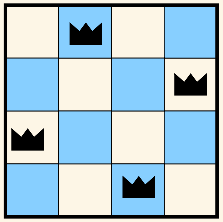
- Suppose A's worst-case time complexity $t_A(n) =_{\text{def}} n^2$ and algorithm B's worst-case time complexity $t_B(n) =_{\text{def}} n!$.





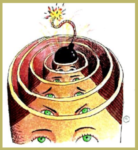
25.3. The n-Queens Problem

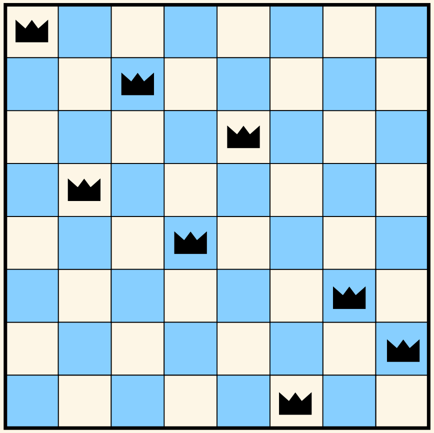
- Can you place 4 Queens on a 4×4 chessboard so that no two can 'take' each other, i.e. no two are on the same row, column or diagonal?
- Here's an answer in the case of 4-Queens:



- Can you find an answer for the 8-Queens Problem?

  This is *not* an answer:



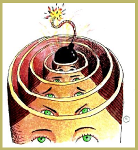


- As they stand, the 4-Queens Problem and the 8-Queens Problems have only one instance (there's no parameters). So, to make matters more interesting, let's generalise the problem.

Problem 25.1. *The n-Queens Problem*

Parameters:	*An integer n, $n \geq 4$.*
Returns:	*All ways to place n Queens on a $n \times n$ chessboard so that no two can 'take' each other.*

- Some observations about solutions:
  - Each Queen must be on a different row.



– Assume Queen i is placed on row i.

– So all we have to do is choose the columns.

– A candidate configuration can be represented by an n-tuple, $\langle x_1, x_2, \ldots, x_n\rangle$, where x_i is the column on which Queen i is placed.

– E.g., the diagram above would be represented as

$$\langle 1, 3, 5, 2, 4, 7, 8, 6\rangle$$

25.3.1. A Naïve Algorithm

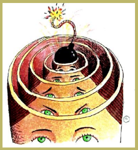
- Here is a possible algorithm for solving the n-Queens Problem.

  In fact, as you'll see when we analyse its complexity, it's a really poor algorithm. So I've deliberately not taken the trouble to even write it out in detail. I've just the sketched the main idea in *very, very* high-level terms.

```
while there are untried configurations
{   generate the next configuration of the $n$ Queens;
    if no two Queens can 'take' each other
    {   print this configuration;
    }
}
```

– Let's count how many configurations the algorithm tests.

– This algorithm *generates-and-tests* every possible tuple $\langle x_1, x_2, \ldots, x_n\rangle$



- E.g. for 4-Queens, it generates

$$\langle 1,1,1,1\rangle, \langle 1,1,1,2\rangle, \langle 1,1,1,3\rangle, \ldots \langle 1,1,2,1\rangle, \langle 1,1,2,2\rangle, \langle 1,1,2,3\rangle, \ldots$$

- x_1 can be any one of n values, so can $x_2, \ldots,$ so can x_n.
- Therefore, it tests n^n configurations.

$$t(n) =_{\mathrm{def}} n^n$$

- $4^4 = 256; 8^8 = 16777216$

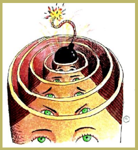
25.3.2. An Improvement

- The naïve algorithm ignores an obvious constraint that can save us a huge amount of effort:

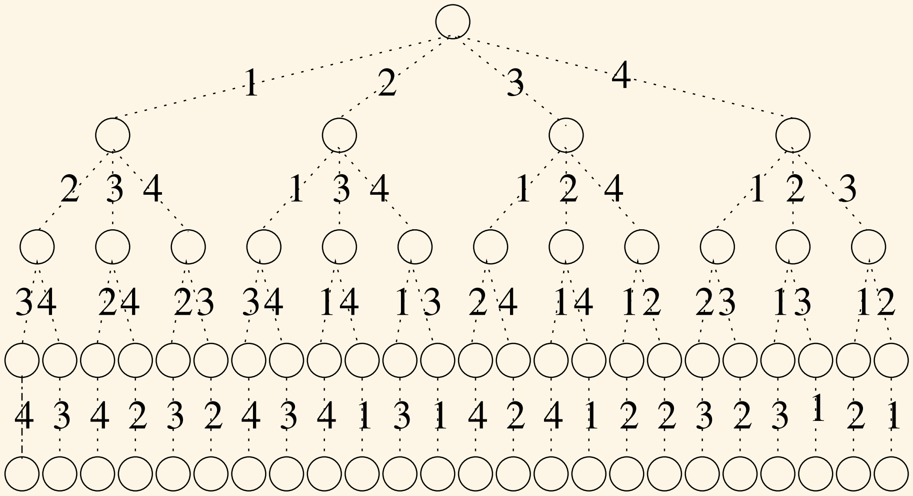
  No two Queens can be in the same column — I.e. no two values in a tuple can be the same.

- Here, still in *very, very* high-level terms is a revised algorithm:

```
while there are untried configurations
{   generate the next configuration of the $n$ Queens...
    allowing no duplicates values in the configuration;
    if no two Queens can 'take' each other
    {   print this configuration;
    }
}
```



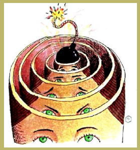
item We can visualise the configurations that this algorithm generates and tests for $n = 4$. The algorithm carries out a test at the end of each branch of this tree:



- You can select any one of n values for x_1.
- You can select any one of $n-1$ values for x_2.
- $\vdots$
- You can select the one remaining value for x_n.
- Therefore, it tests $n!$ configurations.

$$t(n) =_{\text{def}} n!$$

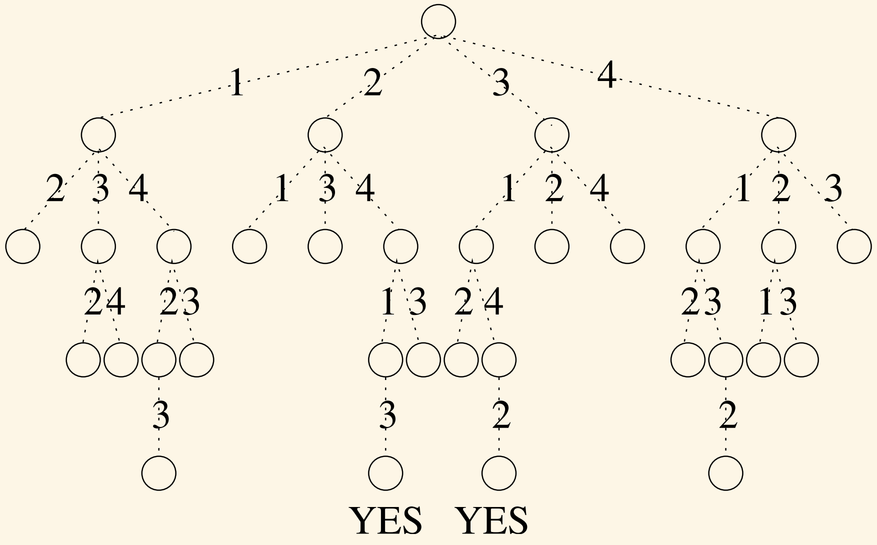
- $4! = 24; 8! = 40320$



25.3.3. Another Improvement

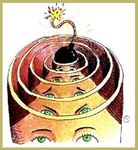
- But, we can test incomplete configurations and abandon them early if they already contain two Queens that can take each other.

  We can visualise this using the following diagram:



- This looks good! So let's see what it looks like in DECAFF. The easiest implementation is a recursive one. There's no need to spend too much time puzzling over how it works. I'm only showing it to you as a matter of interest. We've done what we set out to do, i.e. discuss the complexity.

  Assume x is a global variable and that it is an array of length n. (If $x[k]$ contains i, this means that the kth Queen is in row k and column i.) To start the algorithm,

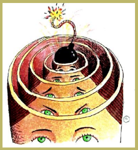


you would call NQueens(1).

Algorithm: NQueens(k)

```
for $i := 1$ upto $n$
{    if CanPlace($i, k$)
     {    $x[k] := i$;
          if $k = n$
          {    print array $x$;
          }
          else
          {    NQueens($k + 1$);
          }
     }
}
```

It uses a procedure that returns a Boolean to say whether Queen i can be placed

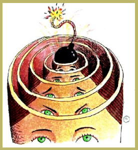


into column k:

```
procedure CanPlace($i, k$)

 for $j := 1$ upto $k - 1$
 {   if $x[j] = i$; // Two in same column
     {   return false;
     }
     if ABS$(x[j] - i)$ = ABS$(j - k)$ // Two in same diagonal
     {   return false;
     }
 }
 return true
```

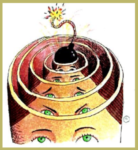
- It's much harder to determine the complexity of this algorithm.
  - On the one hand, it does tests at each point along the paths, not just at the ends of the paths. So that's more work!
  - On the other hand, many paths are abandoned before all n Queens have been placed.
  - What do you think the worst case is?
- There are many further ways of trying to speed up this algorithm, e.g. avoiding the duplicated effort that results from symmetries in the problem.



Acknowledgements

I based some of the n-Queens material on the treatment given in [HSR96].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.



References

[HSR96] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms/C++*. W.H. Freeman, 1996.