https://liuonline.sharepoint.com/sites/Lisam_TDDE25_2021HT_AB

Algorithmics and Computability Part II

TDDE25

Fö 10
Chap 5: Algorithms
Chap 12: Theory of Computation

Patrick Doherty
Dept of Computer and Information Science
Artificial Intelligence and Integrated Computer Systems Division

LINKÖPING UNIVERSITY

Computational Complexity Theory

LINKÖPING UNIVERSITY

Computational Problems

Computational Problems that CANNOT be solved by any algorithm

Computational Problems that CAN be solved by an algorithm

Computability Theory

Computational Problems that CAN be solved by an algorithm

CANNOT be solved in any practical sense due to excessive time/space requirements

CAN be solved in a practical sense with reasonable time/space requirements

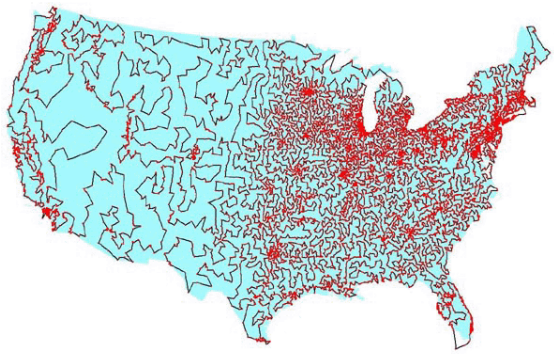
Intractable Class NP

Computational Complexity Theory

Tractable Class P

LINKÖPING UNIVERSITY

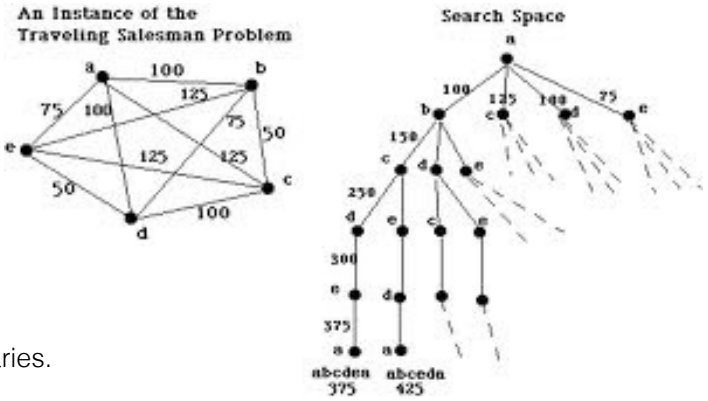
Traveling Salesman Problem



- The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics.
- A traveling salesman has ***n*** number of cities to visit. He wants to know the shortest route which will allow him to visit all cities one time and return to his starting point.
- Solving this problem becomes MUCH harder as the number of cities increases; the figure in the middle shows the solution for the 13,509 cities and towns in the US that have more than 500 residents.
- The TSP problem is in the complexity class *NP-Complete*.

LINKÖPING UNIVERSITY

Traveling Salesman Problem



- Suppose there are *n* cities to visit.
- The number of possible itineraries is *(n-1)!*
  - For *n=10* cities, there are *9!=362,880* itineraries.
- What if *n=40*? For brute force methods:
  - There are now *39!* itineraries to check which is greater then 10^{45}
  - Examining 10^{15} tours per second, the required time would be several billion lifetimes of the universe (universe is estimated to be 13.8 billion years old)
  - In fact, no supercomputer, existing or projected can run this fast.
- There are other methods though (branch-bound, progressive improvement, heuristics) that can compute up to *200* cities in "reasonable" time. Current record (?): *85,900* cities for a problem instance.
- *(n-1)!* grows faster than 2^n. So the time it takes to solve the problem grows exponentially with the size of the input.

LINKÖPING UNIVERSITY

Sequential Search: Telephone Book

- Suppose a telephone book has *N=1000000* entries.
- Given the name *Y*, search the telephone book sequentially for *Y*'s telephone number
  - Entries $\langle X_1, T_1\rangle, \langle X_2, T_2\rangle, \ldots X_{1000000}, T_{1000000}\rangle$
  - At each iteration *Y* is compared with X_i
  - Assume time increases relative to the number of comparisons, so we are counting comparison instructions. (there may be other instructions...)
- In the worst case, *1 000 000* comparisons may have to be made.
- Call the algorithm *A*. We say it has a *worst case* running time which is on the order of *N*.
- *A* runs in time *O(N)* in the worst case, where *N* is the number of entries in the telephone book.
  - In other words, the time complexity of *A* is dependent on the size of the input.
  - *A* has worst case behavior which is *linear in the size* of the input to *A*.

LINKÖPING UNIVERSITY

Big-O Notation

- We do not care whether the algorithm takes time N,$3N$,$100N$, or even a fraction of N: $N/6$
- The only thing that matters is that the running time of the algorithm grows linearly with N.
  - In other words, there is some constant k such that the algorithm runs in time that is no more than $k * N$ in the worst case
- Let $T(n)$ be a function on n (the size of the input to an algorithm) then $T(n)$ is on the order of $f(n)$ if:

$T(n)$ characterizes the running time of the algorithm, i.e, lines of code, # of additions, etc. as a function of input n.

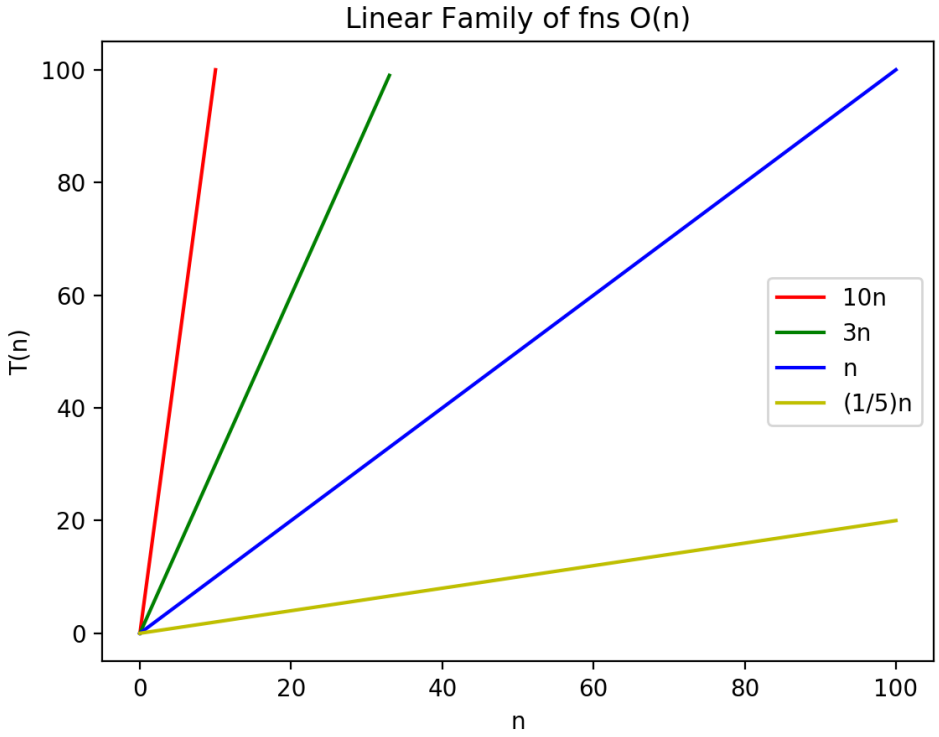
$T(n)$ usually characterizes worst case running time.

Asymptotic Analysis

$T(n)$ is $O(f(n))$ if $T(n) < k * f(n)$ for some k, for all $n > n_0$

LINKÖPING UNIVERSITY

Linear Family of Functions



Multiplicative constants only change the slope of the curve and not its shape.

LINKÖPING UNIVERSITY

Binary Search: Telephone Book

BINARY SEARCH

Best	Average	Worst
O (1)	O (log n)	O (log n)

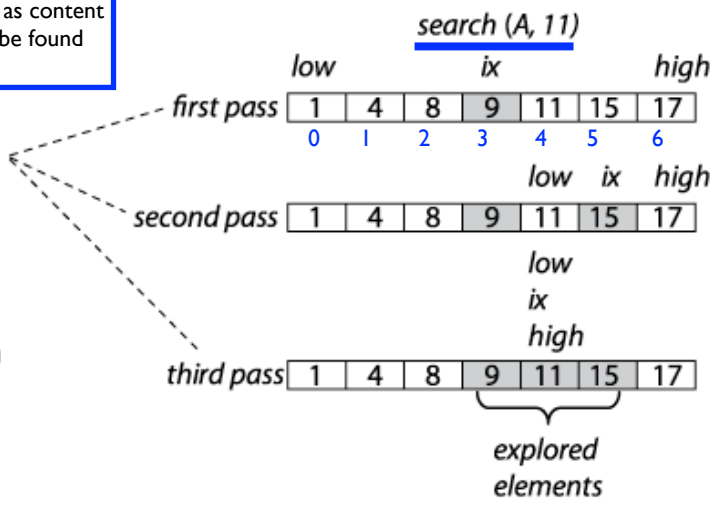
Array

Divide and Conquer

A: Sorted List as content
t: Number to be found
n: size of list

n=7

```
search (A, t)
1.   low = 0
2.   high = n −1
3.   while (low ≤ high) do
4.     ix = (low + high)/2
5.     if (t = A[ix]) then
6.         return true
7.     else if (t < A[ix]) then
8.         high = ix − 1
9.     else low = ix + 1
10.  return false
end
```



LINKÖPING UNIVERSITY

Binary Search Example: Telephone Book

1 Allen
2 Baley
3 Boyer
4 Casy
5 Davis
6 Davison
7 Glen
8 Greer
9 Haley
10 Hanson
11Harrison
12 Listor
13 Mendel
14 Morgenstern
15 Patton
16 Perkins
17 Quinn
18 Reed
19 Schmidt
20 Woolf

Find Casy

Round up on non whole numbers

Casy > Boyer

Eureka!

Casy < Davis

Casy < Davison

Casy < Harrison

Start

Length = 20
# Comparisons = 5

Size of the problem is reduced by half each comparison!

LINKÖPING UNIVERSITY

Binary Search: 1 000 000 Entries

Length = 20
# Comparisons = 5

Length = 1000000
# Comparisons = *??*

- Each comparison reduces the length of the input list by half.
- The process terminates when or before the list becomes empty
- So, the worst case number of comparisons is obtained by figuring out how many times a number N can be repeatedly divided by two before it is reduced to 0 (ignoring fractions)

1000000 is about 2^{20}

$log_2(1000000)$ is about 20

$log_2(N)$ counts the number required to reduce N to 1 (*not* 0)

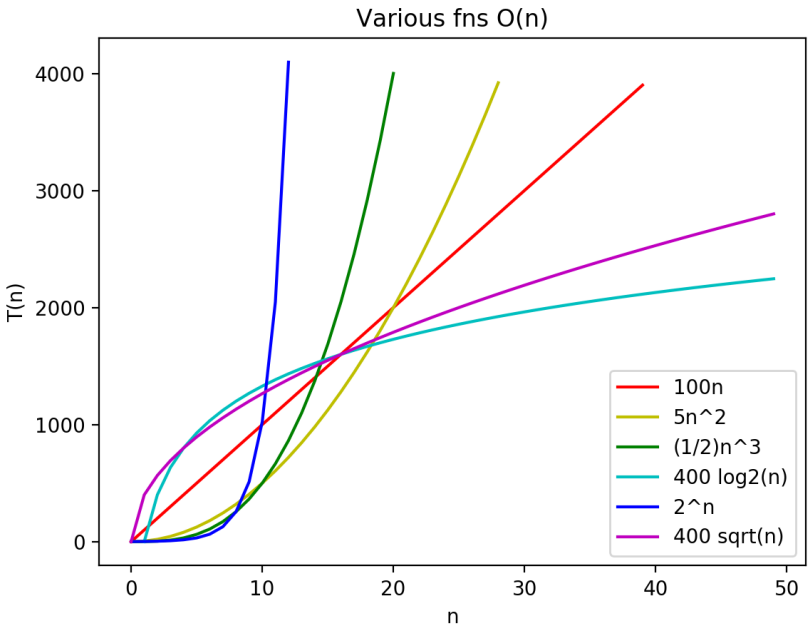
In the worst case, # of comparisons required is $1 + log_2(N)$, or $O(log_2(N))$

Comparing Linear vs. Log

	Sequential search	Binary search
	N	1+ $\log_2$ N
	10	4
	100	7
	1000	10
a million	1000000	20
a billion	1000000000	30
a billion billion	1000000000000000000	60

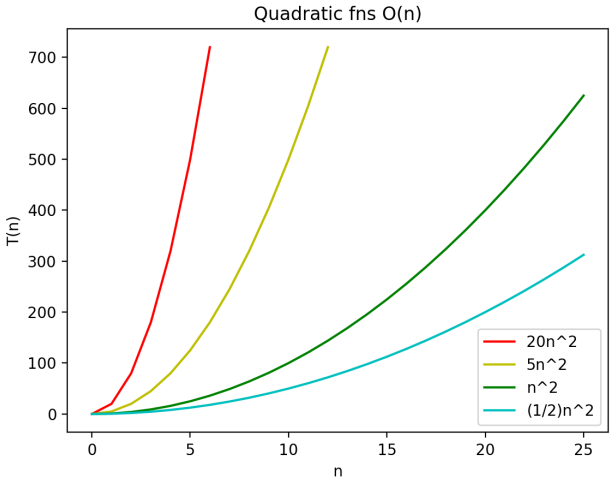
Various Runtimes of different algorithms

Linear	$O(n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Logarithmic	$O(log_2(n))$
Exponential	$O(2^n)$
Square Root	$O(sqrt(n))$



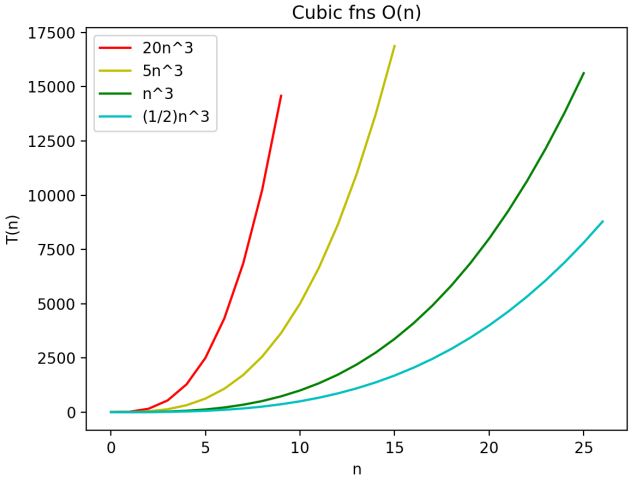
LINKÖPING UNIVERSITY

Family of Quadratic $O(n^2)$/ Cubic $O(n^3)$ Functions



Quadratic

Cubic



LINKÖPING UNIVERSITY

Some Values of some Representative Functions

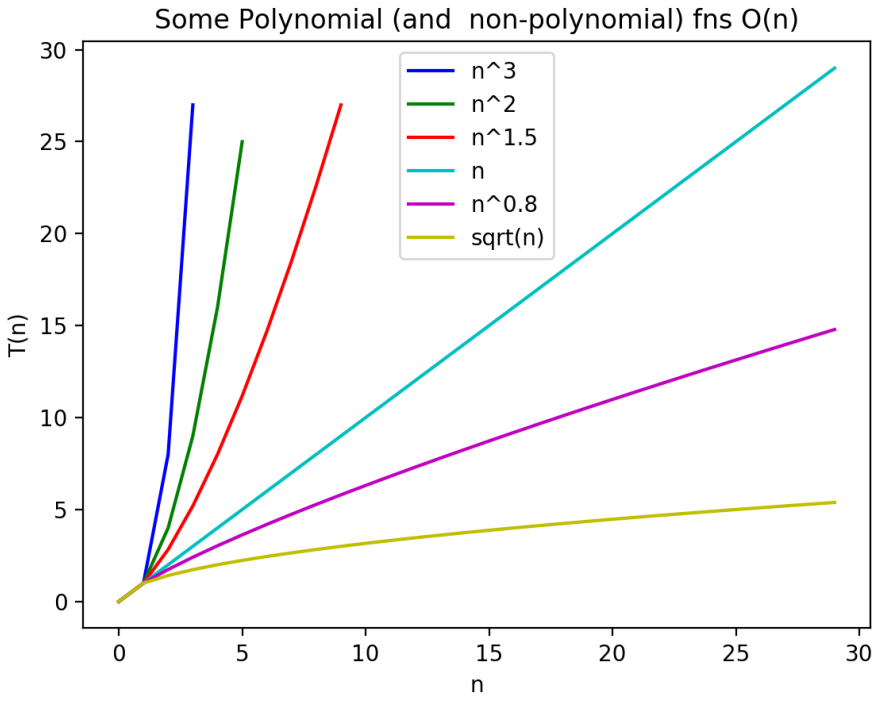
N=	10	50	100	300	1000	
5N	50	250	500	1500	5000	
N x $\log_2$ N	33	282	665	2469	9966	Tractable
N^2	100	2500	10000	90000	1 million (7 digits)	Polynomial
N^3	1000	125000	1 million (7 digits)	27 million (8 digits)	1 billion (10 digits)	
2^N	1024	a 16 digit number	a 31 digit number	a 91 digit number	a 302 digit number	Intractable
N!	3.6 million (7 digits)	a 65 digit number	a 161 digit number	a 623 digit number	unimaginably large	Super Polynomial
N^N	10 billion (11 digits)	an 85 digit number	a 201 digit number	a 744 digit number	unimaginably large	

The number of protons in the known universe has 126 digits

The number of microseconds since the Big Bang has 24 digits

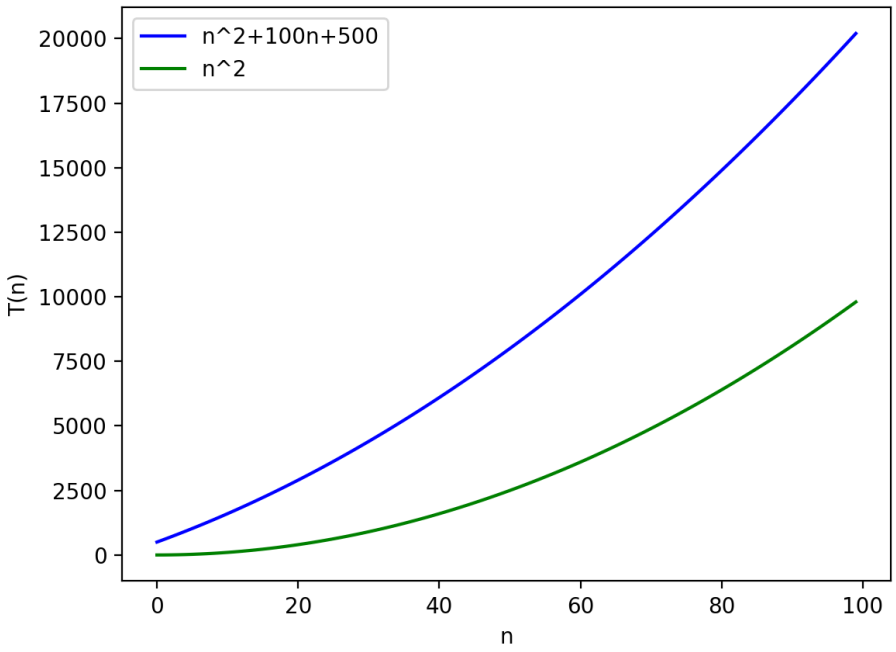
Polynomials

The basic shape of a polynomial function is determined by the highest valued exponent in the polynomial (called the order of the polynomial).

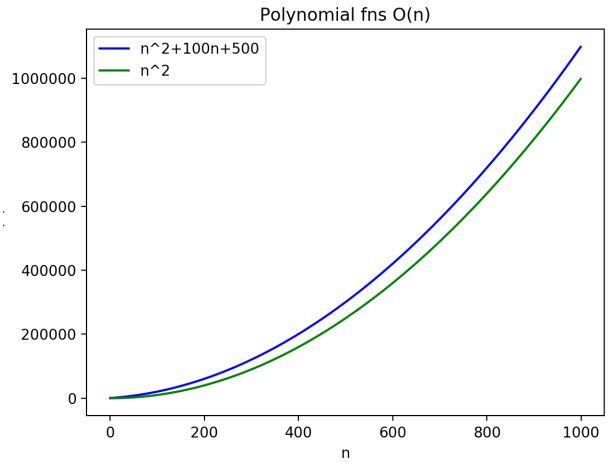


Dominant Terms of a Polynomial ($O(p(n))$

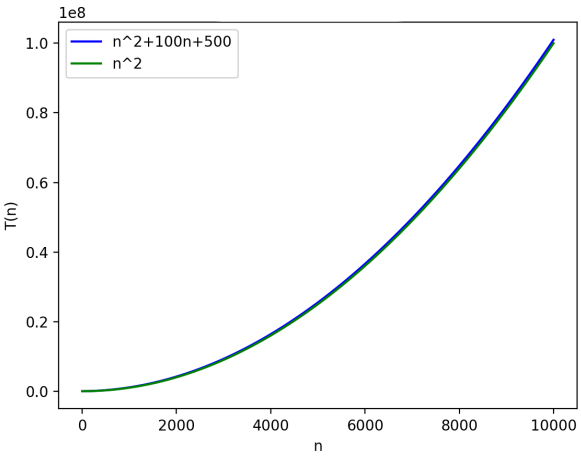
Only the dominant terms of a polynomial matter in the long run.
Lower-order terms fade to insignificance as the problem size increases.



Dominant Terms of a Polynomial ($O(p(n))$

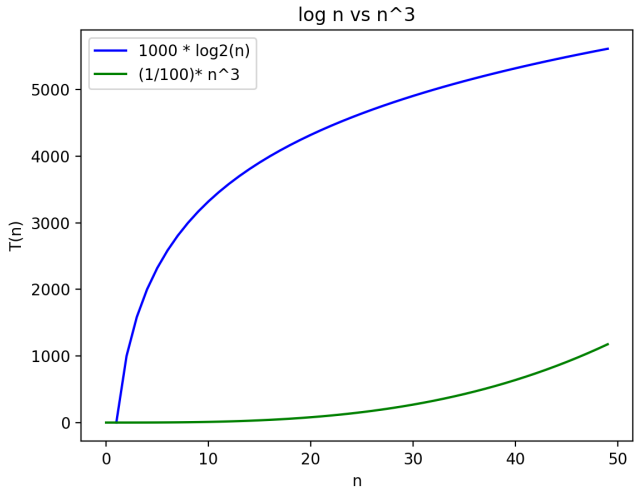


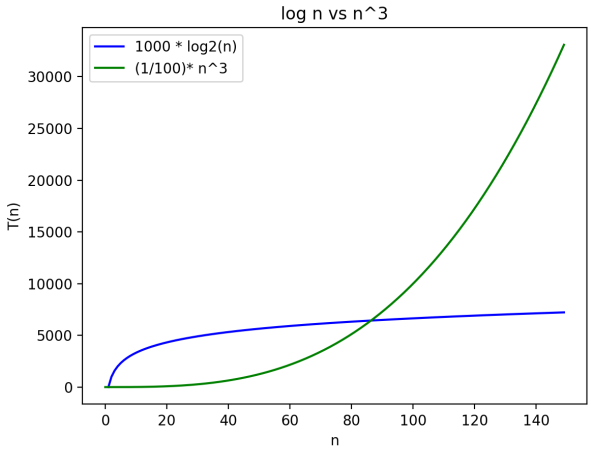
As n grows



$O(log_2(n))$ vs. $O(n^3)$

Polynomial curves will always overtake logarithmic curves eventually, when the problem size gets big enough, regardless of the multiplicative constants involved.

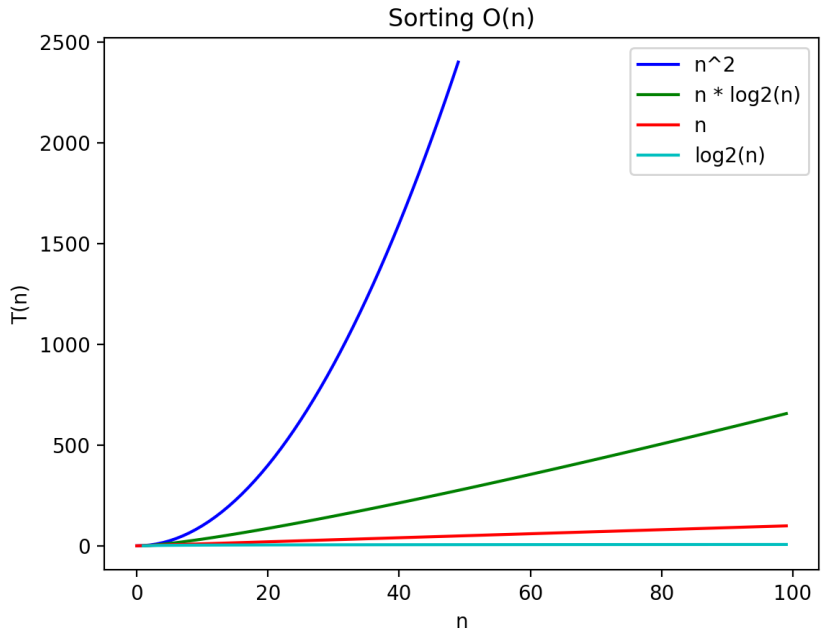




LINKÖPING UNIVERSITY

Sorting Algorithms

The best sorting algorithms (such as mergesort) run in $O(n * log_2(n))$ time. Slower ones (such as bubble sort, selection sort, and insertion sort), take $O(n^2)$ time.



LINKÖPING UNIVERSITY

The Class P of Computational Problems

- A function $f(N)$ is said to be bounded from above by another function $g(N)$, if for all N from a certain point on, $f(N)$ is no greater than $g(N)$.
  - $log_2\ N$ is bounded by N
  - $N * log_2\ N$ is bounded by N^2
  - 2^N is bounded by $N!$
  - $N!$ is bounded by N^N
- For our purposes a polynomial function of N is one that is bounded from above by N^k for some fixed k.
  - All other functions are super polynomial (exponential)
- Recall definition of a polynomial function:
  - is an expression of finite length constructed from variables and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents.
- An algorithm whose order-of-magnitude time performance is bounded from above by a polynomial function of N, where N is the size of its inputs, is called a polynomial time algorithm ($O(p(N))$, where p is a polynomial function)

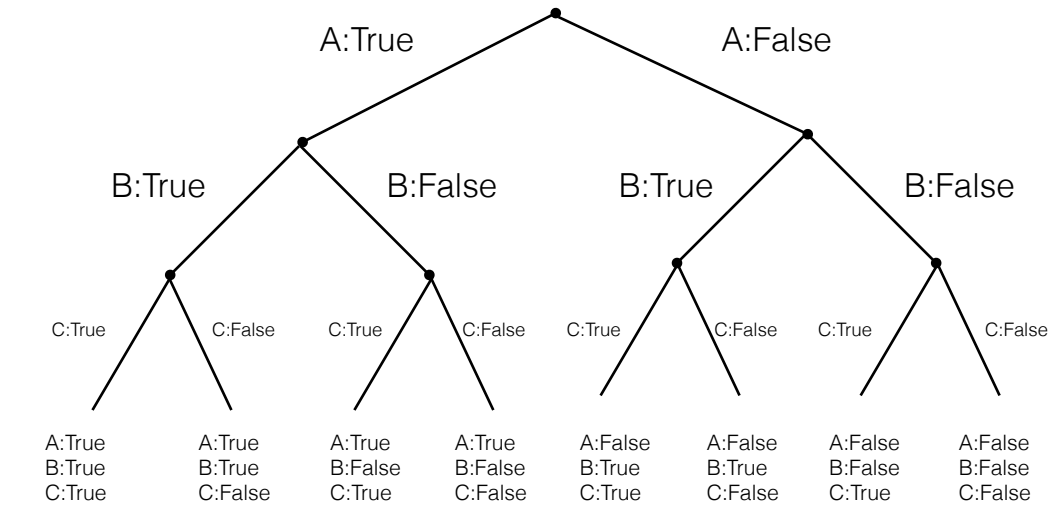
The Class of NP Computational Problems

- Most exponential time algorithms are merely variations on exhaustive search.
  - Generate and test
  - Incrementally generate partial solution and Backtrack
- Some examples of problems:
  - Traveling salesman problem
    - find the shortest path passing through all nodes in a graph only once.
  - shortest path problem
  - 3-coloring map problem
  - Is there a Hamiltonian path in a graph?
    - is there a path passing through all nodes in a graph only once.
  - Satisfiability problem

The Satisfiability Problem

- Satisfiability Problem
  - Find a truth assignment that satisfies a sentence in the propositional calculus
  - Existing algorithms are exponential in the size of the input formula.
    - But! ....
    - If one has a truth assignment for a formula, certifying that it is in fact a valid truth assignment can be checked in polynomial time.
    - Let's call such a solution a "short" certificate (it's size is bounded by a polynomial)

Formula to satisfy: $A \wedge (\neg B \vee C) \wedge (\neg C \vee B)$



Short Certificate

2^3 Models

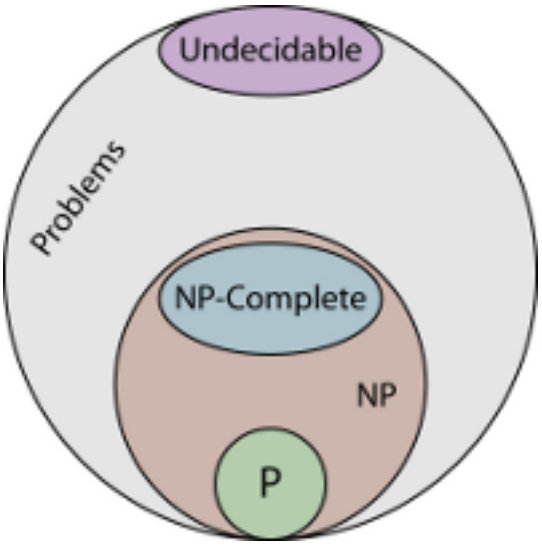
The Class of NP Computational Problems

- Define the following non-deterministic algorithm (for satisfiability checking):
  - Assume there is a *magic coin* that when flipped will always provide the right choice for assigning *T* or *F* to one of the variables in the input formula.
    - If only one of two choices can be extended to a complete solution, the magic coin will choose it (without looking ahead);
    - If both choices can be used, or neither can, it acts like a normal random coin
  - Whenever there is a *choice*, use the magic coin.
- It can be shown that for this class of problems, each has a polynomial time nondeterministic algorithm!
  - It is proved by showing that the "short" certificates discussed previously correspond to the polynomial time "magical" executions.
  - So simply follow the instructions of the magic coin and when there is a complete candidate solution simply check whether it is legal.
    - Since the coin chooses the best possibility, a no is a no and a yes is a yes and this can be checked in polynomial time.

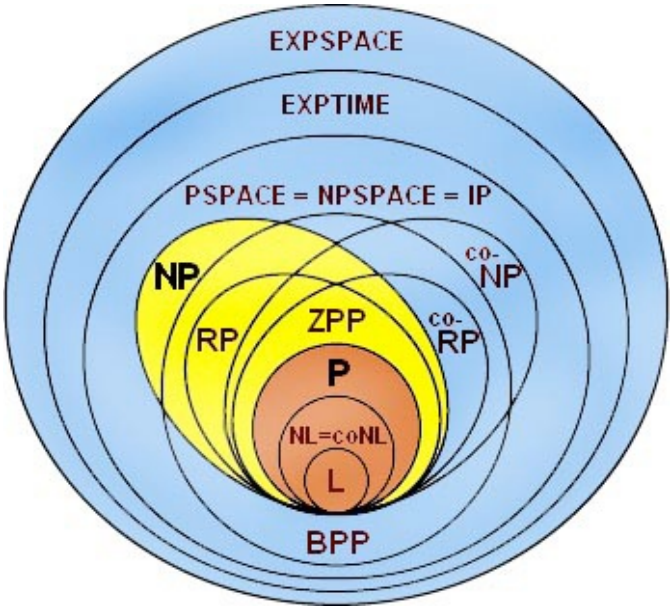
$P = NP?$

- P stands for the class of problems that admit polynomial time solutions
- NP stands for the class of problems that admit nondeterministic polynomial time solutions
- The *NP-complete* problems are the hardest problems in the class of NP problems
  - If one of them turns out to be "easy", that is it is in P, then all NP problems are in P.
  - The SAT problem and the Traveling Saleman Problem are *NP-complete problems.*
- Since P is already a part of NP, an important unresolved question is whether $P = NP$?
  - This has been an open problem since it was posed in 1971
  - It is one of the most difficult unresolved problems in computer science and mathematics
  - Most computer scientists believe P is not equal to NP

Complexity Classes



Computational Problems



Complexity Zoo!

Over 500 classes to date!

LINKÖPING UNIVERSITY

Prime Numbers

Fundamental Theorem of Arithmetic

Every integer greater than 1 is either a prime number or the product of prime numbers.

also called the unique prime factorization theorem

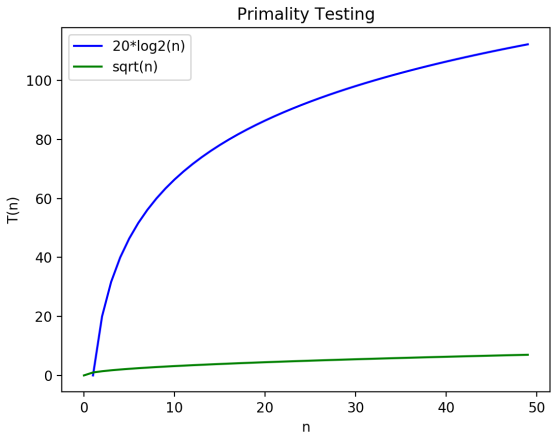
Testing whether a number is Prime is now known to be in P

Generating the prime factorization of a composite number is currently in NP

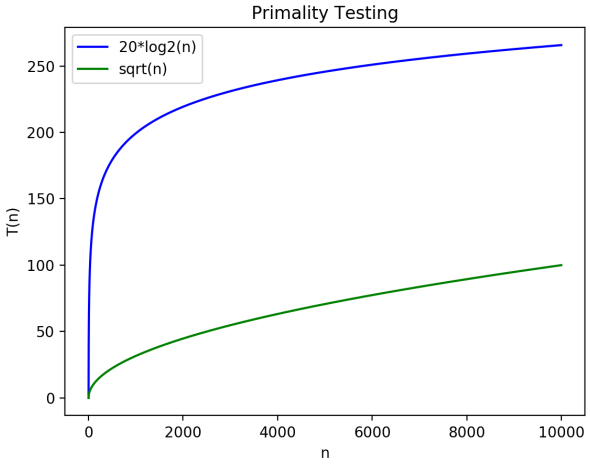
LINKÖPING UNIVERSITY

Primality Testing

The superiority of the $O(log_2(n))$ Fermat prime test over the $O(sqrt(n))$ prime test becomes clear for really big integers.



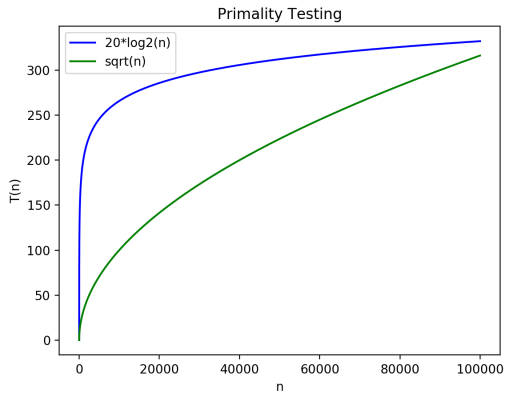
Testing whether a number is Prime or not is in class P. Providing prime factors for a number is currently known to be in NP, but it is not known if it is NP-Complete.

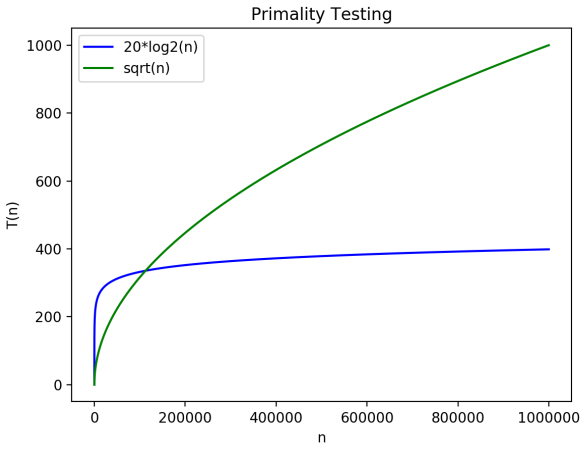


LINKÖPING UNIVERSITY

Primality Testing

The superiority of the $O(log_2(n))$ Fermat prime test over the $O(sqrt(n))$ prime test becomes clear for really big integers.





Testing whether a number is Prime or not is in class P.

Providing prime factors for a number is currently known to be in NP, but it is not known if it is NP-Complete (probably not)

LINKÖPING UNIVERSITY

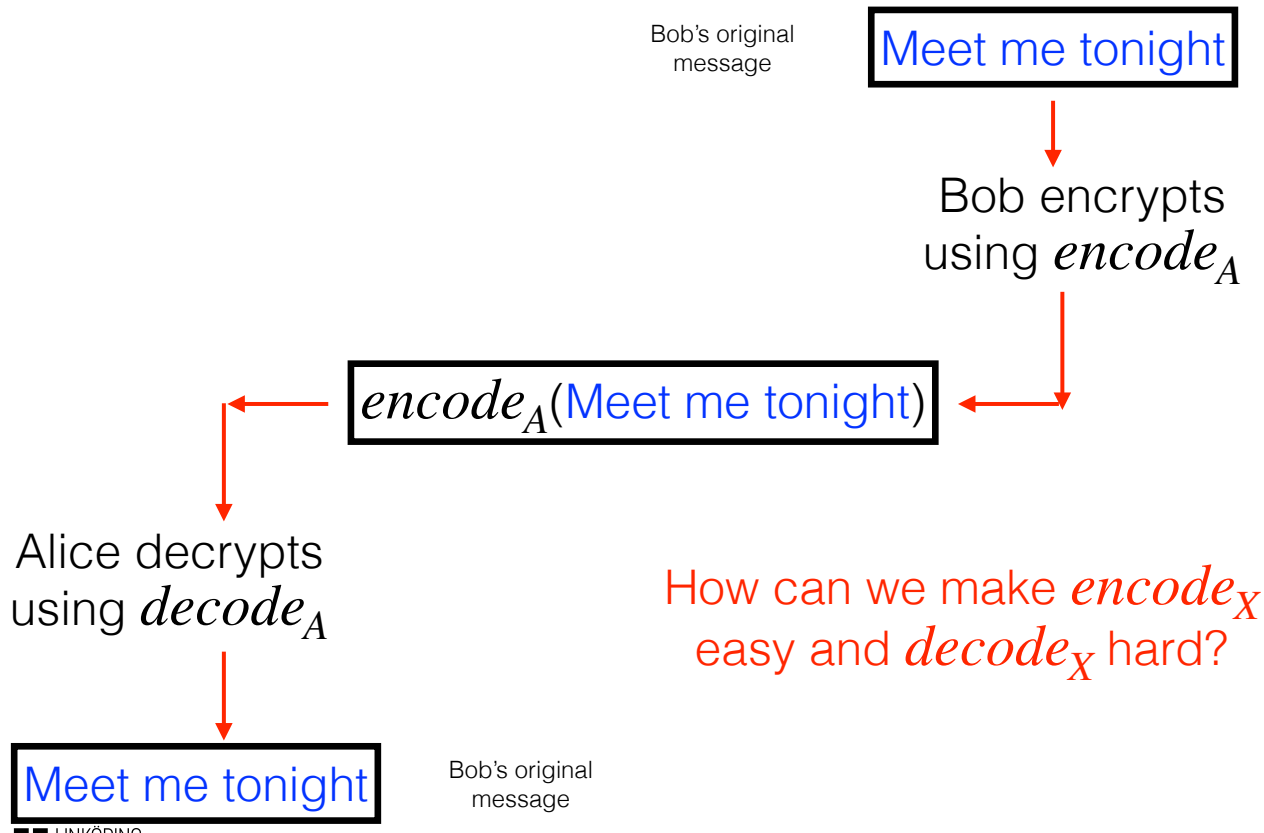
Cryptography

- Basic activities in cryptography are to
  - *Encode* (Encrypt) a message
  - *Decode* (Decrypt) a message
- Done in such a way that the recipient can decode it, but eavesdroppers can not.
- Two procedures of interest:
  - $M^* = encode(M)$
  - $M = decode(M^*)$

Public-Key Cryptography

- Proposed by Diffie and Hellman (1976)
  - Idea is basis for the RSA Algorithm (Rivest, Shamir, Adelman)
- Messages to be sent are assumed to be sequences of digits.
- Given a particular Party A,
  - A's encode procedure $encode_A$ is associated with a public key, known to anyone who wants to send a message to A, and
  - A's decode procedure $decode_A$ is associated with a private key only known to A
- Each party has its own *Public* and *Private Keys*.

The Process: $decode_A(encode_A(M)) = M$



RSA Algorithm

- The RSA system is based on the contrast between testing a number for primality and factoring a number.
  - Generating and Testing for primality can be done fast using probabilistic algorithms. Testing is in class P
  - Prime Factoring of a number is conjectured to be in class NP and there are no known algorithms to do this efficiently.

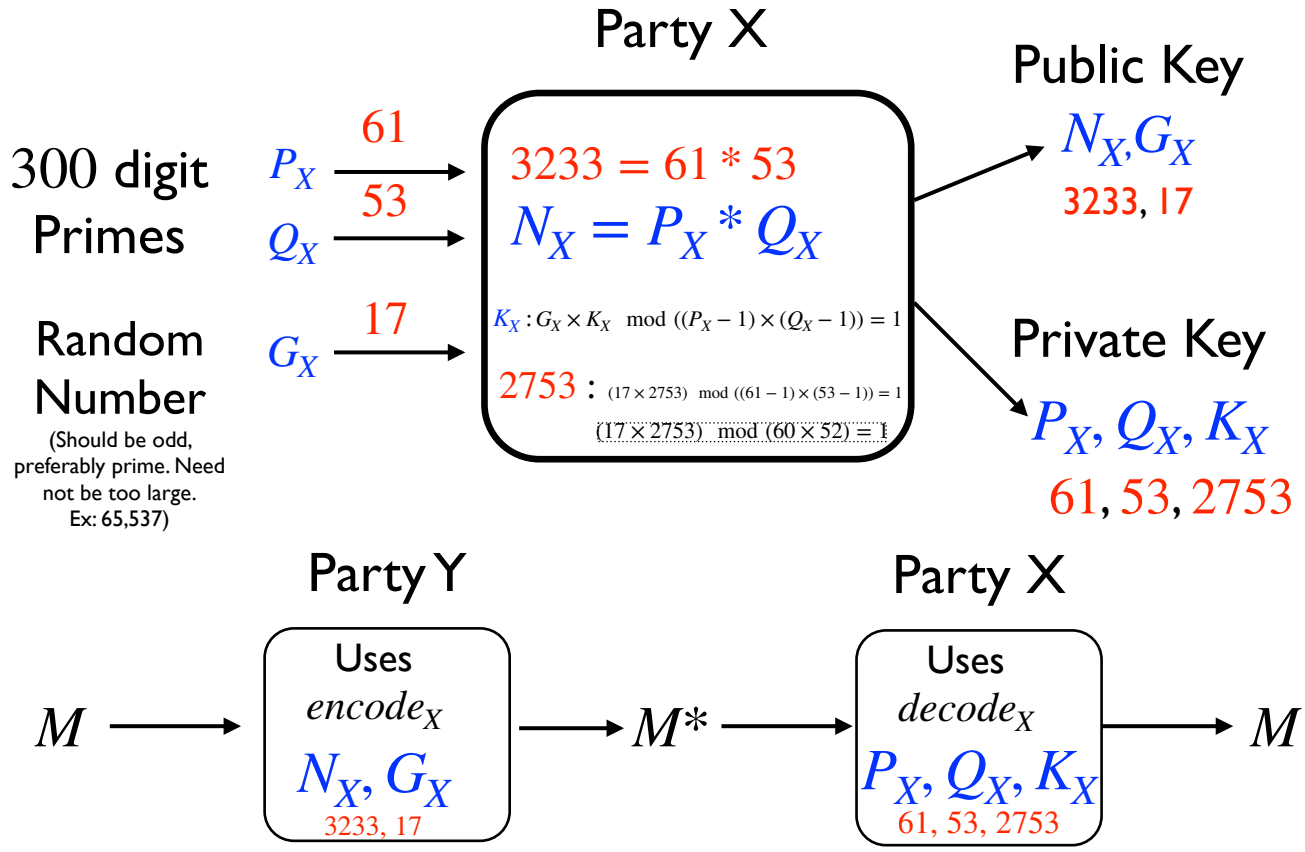
Basics of RSA

- (Each) party X, secretly and at random, chooses two large prime numbers P and Q, of length around 200 digits, and multiplies them: $N = P * Q$.
- Party X then chooses a relatively large random number G, say $65537 = 2^{16} + 1$ (public exponent)
  - G should have no common factors with the product of $(P-1)*(Q-1)$ (except for 1)
- Finally, party X computes the number K (private exponent) to be the modular multiplicative inverse (G^{-1}) of $G \ mod \ (P-1)*(Q-1)$:
  - Find G^{-1} such that $G \times G^{-1} \equiv 1 \mod ((P-1) \times (Q-1))$
  - This means that $G * K$ yields a remainder of 1 when divided by $(P-1)*(Q-1)$:
    - $G \times K \equiv 1 \mod ((P-1) \times (Q-1))$
    - $(G \times K) \mod ((P-1) \times (Q-1)) = 1$

mod: remainder after division
x mod y = r

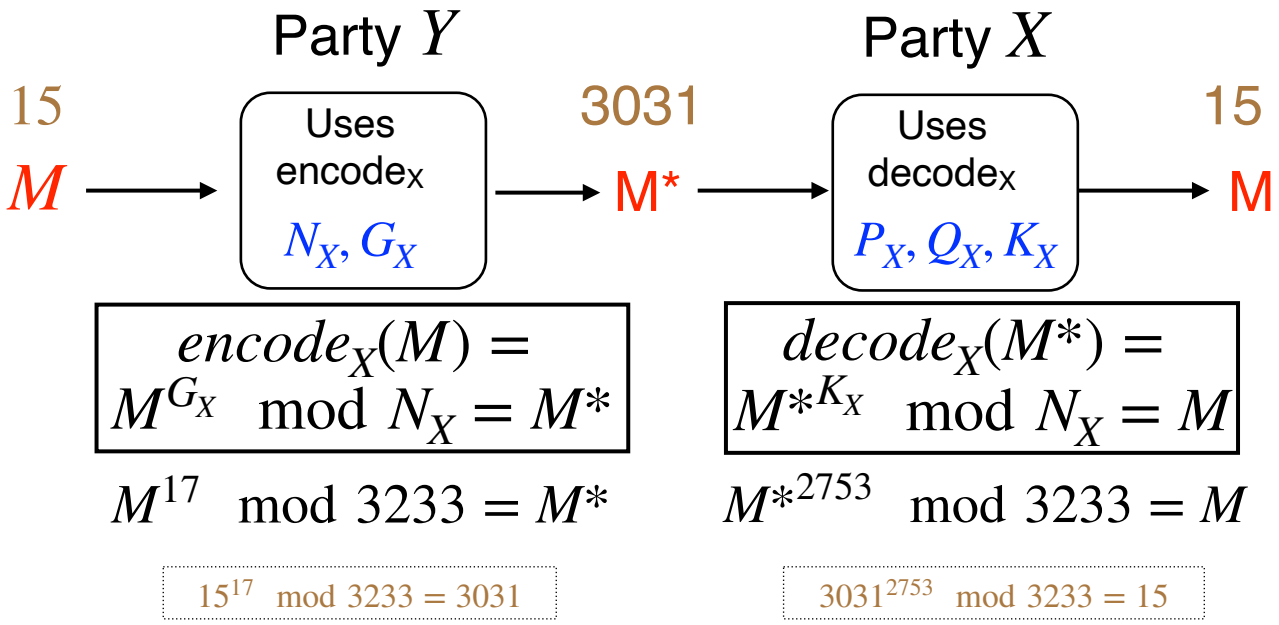
LINKÖPING UNIVERSITY

Basic Idea



LINKÖPING UNIVERSITY

Encoding and Decoding



$$encode_X(M) = M^{G_X} \mod N_X = M*$$

$$decode_X(M*) = M*^{K_X} \mod N_X = M$$

$$M^{17} \mod 3233 = M*$$

$$M*^{2753} \mod 3233 = M$$

$15^{17} \mod 3233 = 3031$

$3031^{2753} \mod 3233 = 15$

If an adversary Z could factor large numbers efficiently:
Since N_X is public, it could be factored into P_X, Q_X
Since G_X is public, P_X, Q_X, N_X, G_X could be used to compute K_X the secret key!

LINKÖPING UNIVERSITY

Digital Signatures

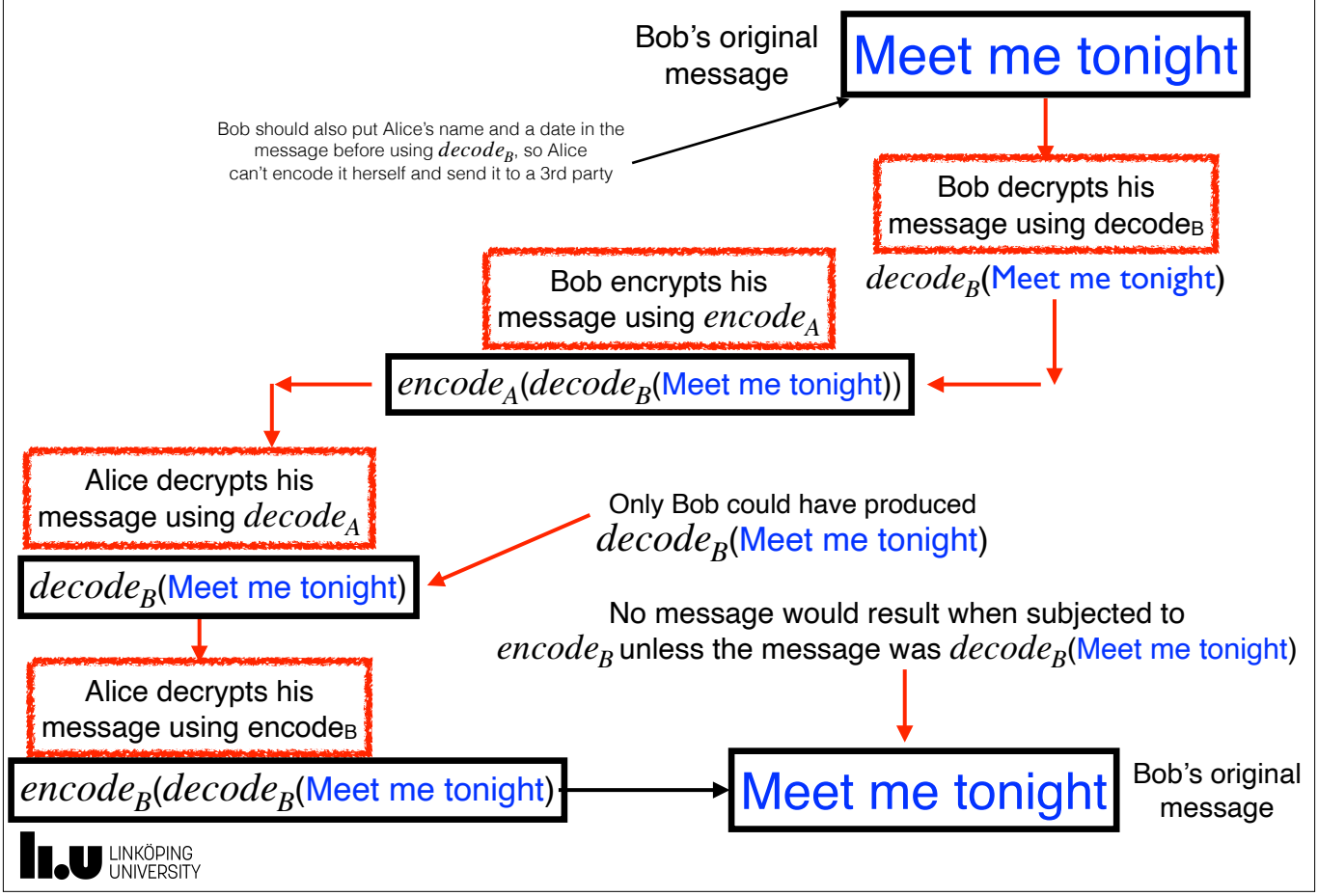
- Can a message be "signed" by the sender, so that:
  - the receiver can be sure that the sender alone would have sent it
  - the sender can not later deny sending a message
  - and, the receiver having received the message, can not sign any message in the sender's name?
- Digital signatures are important for money orders, electronic contracts, business transactions, etc.
- The Main idea: Make the *encode* and *decode* functions commutative

$$decode_X(encode_X(M)) = M \qquad encode_X(decode_X(M)) = M$$

$$M = M^{G_X * K_X} (mod\ N_X)$$

LINKÖPING UNIVERSITY

The Process: Signed Encrypted Message



Conclusions on RSA

- All approaches suggested so far in attempting to break the RSA system have been shown to require fast solutions to the factoring problem too.
  - Since Factoring is strongly conjectured to have no fast algorithm, not even a probabilistic algorithm, RSA is considered safe under certain assumptions!
- The size of the number (# of digits) should be large!
  - *1999*: a 140 digit number was factored using several hundred computers running for several months.
  - *2000*: An optical device called Twinkle (not built but specified) would be able to factor 160 digit numbers in a few days if 12 were used.
    - RSA used 512 bit numbers (between 154 -155 digit numbers)
      - 512 bit numbers are no longer considered safe!
  - *2020:* The largest publicly known factored RSA number was 829 bits (250 decimal digits).
    - It was factored by a state-of-art distributed implementation taking 2700 CPU years.
    - Minimum recommendations for keys are now 2048 bits.
- Slightly different version of the RSA system (Rabin) exists whose security is provably equivalent to fast factoring.
  - good to have if there are other ways to break the system that do not appear to require fast factoring. After analysis, they will be shown to require ff.

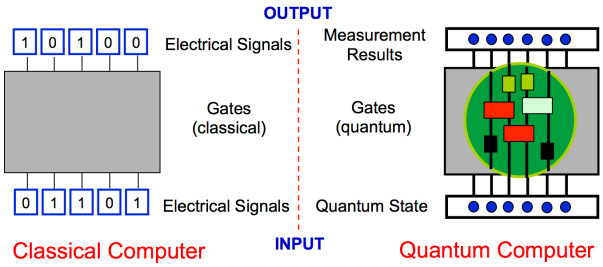
Quantum Computing

- If a computer could be built based on the the laws of quantum physics rather than classical physics, one might get an exponential speedup for some types of computations
  - Quantum analog of a bit is a qubit
    - the direction of photon polarization (vertical or horizontal)
    - nuclear spin (a special two-valued quantum observable)
    - energy level of an atom (ground or excited)
  - Basis states of a qubit: $| 0 > \text{ or } | 1 >$
    - a qubit can be in both states simultaneously with certain probabilities (which can even be negative or imaginary)
    - the resulting combination state is a superposition
      - two basis states at the same time.
      - but once we take a look, all probabilities disappear and it is in one state or another.
  - Combination states of qubits: $| 0101 >$
    - Some can be entangled: observing one and fixing its state causes the other to lock into its state simultaneously: instant communication!

Quantum Computing cont'd

- Full, general purpose quantum computing subsumes classical computation
  - can emulate any classical computation without loss of time
- A classical computer can simulate any quantum computation
  - but it may result in an exponential loss of time
- Church/Turing Thesis remains intact
  - Quantum computing can only compute computable functions.
- Is there a computational problem with exponential lower bound in classical models of computation that has a polynomial-time quantum algorithm?

Big Surprise for the Future



- SHOR's Algorithm
  - Invented by Peter Shor in 1994
  - Quantum algorithm for integer factorization:
    - Given an integer N, find its prime factors.
    - Theoretically, Shor's algorithm runs in polynomial time $O((log_2(N))^3)$ on a quantum computer, (if one can build a quantum computer!)
- The algorithm requires a huge amount of quantum gates: 4096-bit number requires roughly $5 * 10^{12}$ (4,947,802,324,992) quantum gates.
  - Gates required increase with N as $(log_2(N))^3$.
  - *2012*: factorization of 21 succeeded (10 qubits required) on a quantum computer (simulator?)
    - needed prior knowledge of the solution
  - *2012*: factorization of 143 succeeded (4 qubits required) using a new algorithm: minimization.
    - did not need prior knowledge of the solution. (simulator?)
  - 2020: factorisation of 1028171 using quantum annealing (D-Wave)
    - Algorithm is different from Shor's algorithm
    - D-Wave's is a "quantum annealer" which is more of a simulator than a computer. (2000Q: 2000 qubits)