Algorithm runtime analysis and computational tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage

Charles Babbage (1864)  Analytic Engine (schematic)
Time Complexity of an Algorithm

How do we measure the **complexity** (time, space requirements) of an algorithm.

The **size** of the problem: an integer $n$
- # inputs (for sorting problem)
- # digits of input (for the primality problem)
- sometimes more than one integer
  - Knapsack problem:
    - $C$: capacity of the knapsack,
    - $N$: number of objects

We want to characterize the running time of an algorithm for increasing problem sizes by a function $T(n)$
Units of time

1 microsecond?

1 machine instruction?

# of code fragments that take constant time?
Units of time

1 microsecond?

no, too specific and machine dependent

1 machine instruction?

no, still too specific and machine dependent

# of code fragments that take constant time?

yes

# what kind of instructions take constant time?

arithmetic op, memory access, finite combination of these
unit of space

bit?

int?
unit of space

bit?
  very detailed but sometimes necessary

int?
  nicer, but dangerous: we can code a whole program or array (or disk) in one arbitrary sized int, so we have to be careful with space analysis (take value ranges into account when needed). Better to think in terms of machine words

i.e. fixed size, e.g. 64, collections of bits
Worst-Case Analysis

Worst case running time.

A bound on largest possible running time of algorithm on inputs of size n.

- Generally captures efficiency in practice, but can be an overestimate.

Same for worst case space complexity
Average case running time. A bound on the average running time of algorithm on random inputs as a function of input size $n$. In other words: the expected number of steps an algorithm takes.

$$\sum_{i \in I_n} P_i \cdot C_i$$

- $P_i$: probability input $i$ occurs
- $C_i$: complexity given input $i$
- $I_n$: all possible inputs of size $n$

- Hard to model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
- Often hard to compute.
Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
A definition of tractability: Polynomial-Time

**Brute force.** For many problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes exponential time for inputs of size n.
- Unacceptable in practice.
  - Permutations, TSP

An algorithm is said to be polynomial if there exist constants $c > 0$ and $d > 0$ such that on every input of size $n$, its running time is bounded by $c n^d$ steps.

- What about an $n \log n$ algorithm?
Worst-Case Polynomial-Time

On the one hand:
- Possible objection: Although $6.02 \times 10^{23} \times n^{20}$ is technically poly-time, it would be useless in practice, (e.g. n=10)
- In practice, the poly-time algorithms that people develop typically have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

On the other:
- Some exponential-time (or worse) algorithms are widely used because the worst-case (exponential) instances seem to be rare.
  - simplex method solving linear programming problems
Characterizing algorithm running times

Suppose that algorithm A has a running time bounded by

\[ T(n) = 1.62 \, n^2 + 3.5 \, n + 8 \]

- There is more detail than is useful
  - We want to quantify running time in a way that will allow us to identify broad classes of algorithms

- I.e., we only care about Orders of Magnitude and only consider the leading factor
  - in this case: \( T(n) = O(n^2) \)
Asymptotic Growth Rates
Upper bounds

Recap from cs220:

T(n) is \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that

\[
T(n) \leq c \cdot f(n)
\]

for all \( n \geq n_0 \).

Example: \( T(n) = 32n^2 + 16n + 32 \).

- \( T(n) \) is \( O(n^2) \)
- BUT ALSO: \( T(n) \) is \( O(n^3) \), \( T(n) \) is \( O(2^n) \).

There are many possible upper bounds for one function! We always look for the best (lowest) upper bound.
Big O doesn’t always express what we want:

Any comparison-based sorting algorithm requires at least \( c(n \log n) \) comparisons, for some constant \( c \).

- Use \( \Omega \) for lower bounds.

\( T(n) \) is \( \Omega(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 : T(n) \geq c \cdot f(n) \)

Example: \( T(n) = 32n^2 + 16n + 32 \).

- \( T(n) \) is \( \Omega(n^2) \)
Tight Bounds

$T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

**Example:** $T(n) = 32n^2 + 17n + 32$.
- $T(n)$ is $\Theta(n^2)$.

If we show that the running time of an algorithm is $\Theta(f(n))$, we have closed the problem and found a bound for the problem and its algorithm solving it.
$F(n)$ is $O(G(n))$

$F(n)$ is $\Omega(H(n))$

if $G(n) = c \cdot H(n)$
then $F(n)$ is $\Theta(G(n))$

These measures were introduced in CS220
**priority Queue**: data structure that maintains a set $S$ of elements.

Each element $v$ in $S$ has a key $\text{key}(v)$ that denotes the priority of $v$.

Priority Queue provides support for inserting, deleting elements, selection / extraction of smallest (Min prioQ) or largest (Max prioQ) key element, changing key value.
Applications

E.g. used in managing real time events where we want to get the earliest next event and events are added and deleted on the fly.

Also used in Sorting:
- build a prioQ
- Iteratively extract the smallest element

PrioQs can be implemented using heaps
Heaps

Heap: array representation of a

**Complete Binary tree**
- every level is completely filled except the bottom level: filled from left to right
- Can compute the index of parent and children: WHY?
  - parent(i) = floor((i-1)/2)
  - leftChild(i) = 2i+1
  - rightChild(i) = 2(i+1)

Max Heap property:
for all nodes i>0: A[parent(i)] >= A[i]
Max heaps have the max at the root

Min heaps have the min at the root
Heapify\((A,i,n)\)

To create a heap at index \(i\), assuming \(\text{left}(i)\) and \(\text{right}(i)\) are heaps, **bubble** \(A[i]\) **down**: swap with max child until heap property holds

\[
\text{heapify}(A,i,n):
\]

# precondition
# \(n\) is the size of the heap
# tree left\((i)\) and tree right\((i)\) are heaps

....... 

# postcondition: tree \(A[i]\) is a heap
Swapping Down

Swapping down enforces (max) heap property at the swap location:

\[
\begin{array}{c}
\text{new} \\
/ \ \\
y \ \\
\quad x \\
\end{array}
\rightarrow
\begin{array}{c}
\text{x} \\
/ \ \\
y \ \\
\quad \text{new}
\end{array}
\]

\[
\text{new} < x \quad \text{and} \quad y < x:
\]

\[
\Rightarrow \quad x > y \quad \text{and} \quad x > \text{new}
\]

\[
\text{swap}(x, \text{new})
\]

Are we done now?

NO! When we have swapped we need to carry on checking whether new is in heap position. We stop when that is the case.
Heap Extract

Heap extract:
Delete (and return) the root

**Step 1:** replace root with last array element to keep completeness

**Step 2:** reinstate the heap property
Which element does **not** necessarily have the heap property?

How can it be fixed? Complexity?
heapify the root O(log n)

Swap **down:** swap with maximum (maxheap), minimum (minheap) child as necessary, until in place.
Sometimes called bubble down

Correctness based on the fact that we started with a heap, so the children of the root are heaps
Heap Insert

**Step 1:** put a new value into first open position (maintaining completeness), i.e. at the end of the array, but now we potentially violated the heap property, so:

**Step 2:** bubble up

- Re-enforcing the heap property
  - Swap with parent, if new value > parent, until in the right place.
  - The heap property holds for the tree below the new value, when swapping up. **WHY? We only compared the new element to the parent, not to the sibling!**
Swapping up enforces heap property for the sub tree below the new, inserted value:

\[
\begin{array}{c}
  x \\
  \ \ \ \ \ \ \\
  Y \ \ \ \ \ \ \\
  \ \ \ \ \ \ \\
  new
\end{array}
\quad \quad \quad
\begin{array}{c}
  new \\
  \ \ \ \ \ \ \\
  y \ \ \ \ \ \ \\
  \ \ \ \ \ \ \\
  x
\end{array}
\]

if \((\text{new} > x)\) swap\((x,\text{new})\) \\
\(x>y\), therefore \(\text{new} > y\)
Building a heap

heapify performs at most $\lg n$ swaps

why? what is $n$?

buildheap: builds a heap out of an array:

- the leaves are all heaps WHY?
- heapify backwards starting at last internal node

WHY backwards?
WHY last internal node?
which node is that?
LET'S DO THE BUILDHEAP!

[4, 8, 7, 2, 14, 1]
NOW LET'S ADD 18

Build heap [4, 8, 7, 2, 14, 1]  
add 18  
bubble up (2*)
NOW LET'S EXTRACT MAX

Delete max (18)  move 7 to root  bubble down (1*)
Complexity buildheap

Initial thought: $O(n \log n)$, but

half of the heaps are height 0
quarter are height 1
only one is height $\log n$

It turns out that $O(n \log n)$ is not tight!
complexity buildheap

height

0

1

2

3

max #swaps?
complexity buildheap

max #swaps, see a pattern?
(What kind of growth function do you expect?)

<table>
<thead>
<tr>
<th>height</th>
<th>max #swaps, see a pattern?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2*1+2 = 4</td>
</tr>
<tr>
<td>3</td>
<td>2*4+3 = 11</td>
</tr>
<tr>
<td>height</td>
<td>max # swaps</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>0 = 2^1 - 2</td>
</tr>
<tr>
<td>1</td>
<td>1 = 2^2 - 3</td>
</tr>
<tr>
<td>2</td>
<td>2*1 + 2 = 4 = 2^3 - 4</td>
</tr>
<tr>
<td>3</td>
<td>2*4 + 3 = 11 = 2^4 - 5</td>
</tr>
</tbody>
</table>
Conjecture:
height = \( h \)
max #swaps = \( 2^{h+1}-(h+2) \)

Proof: induction
base?
step:
height = \( (h+1) \)
max #swaps:
\[
2\times(2^{h+1}-(h+2))+(h+1)
\]
\[
= 2^{h+2}-2h-4+h+1
\]
\[
= 2^{h+2}-(h+3)
\]
\[
= 2^{(h+1)+1}-(((h+1)+2))
\]

n nodes \( \to \Theta(n) \) swaps
Describe the Heap Operations

Heapify(A,i,n)
A is a heap BELOW i, bubble A[i] DOWN, A is heap at i

Extract(A)
heap not empty, remove and return top, replace by last element,
Heapify top (0), A is a heap, 1 smaller

Insert(A,x)
A is a heap, append x to A, bubble x UP until in place, A is a heap,
1 larger

Buildheap(A)
A is an array, view as complete tree, GO BACKWARDS from last internal, heapifying until top is heapified, now A is a heap
Heapsort, complexity

heapsort(A):
   buildheap(A)
   for i = n-1 downto 1 :
      # put max at end array
      # max is removed from heap
      n=n-1
      #heap one smaller, sorted array one larger
      # reinstate heap property
      # for reduced n elements

- heapify: $\Theta(lgn)$
- heapExtract: $\Theta(lg\ n)$
- buildheap: $\Theta(n)$
- heapsort: $\Theta(n \ lg\ n)$
- space: in place: $\Theta(n)$
DO THE HEAPSORT, DO IT, DO IT!

```
2 4 1
8

1

4 7 8 14
```

```
2
4
1

1

8 14
```

```
7
8
14
```

```
2
4
7
8
14
```

```
1
2
4
7
8
14
```
How not to heapExtract, heapInsert

# These "snail" implementations are NOT preserving the algorithm
# complexity of extractMin: log n and insert: log n and are therefore
# INCORRECT! from a complexity point of view (even though they are
# functionally correct). Remember one of the goals of our course:
# implementing the algorithms maintaining the analyzed complexity
# What are their complexities?

def snailExtractMin(A):
    n = len(A)
    if n == 0:
        return None
    min = A[0]
    A[0]=A[n-1]
    A.pop()
    buildHeap(A)  # O(n)
    return min

def snailInsert(A,v):
    A.append(v)
    buildHeap(A)  # O(n)
Additive Theorem:

Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \). Then \( (f_1 + f_2)(x) \) is \( O(\max(g_1(x), g_2(x))) \).

Sequences of code are additive in complexity:

```c
int c = 0;
// loop 1
for(int i=0; i<n; i++)
    c++;
// loop 2
for(int j=0; j<n*n; j++)
    c++;
```

**Complexity?**

- loop 1: \( O(n) \)
- loop 2: \( O(n^2) \)

**Total complexity:** \( O(n^2) \)
Combinations of functions /code fragments

Multiplicative Theorem:

Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \).
Then \( (f_1f_2)(x) \) is \( O(g_1(x)g_2(x)) \).

Nested code is multiplicative in complexity

\[
\begin{align*}
\text{for} & \ (\text{int } i=0; i<n; i++) \\
& \ \ \ \text{for} (\text{int } j=0; j<n; j++) \\
& \ \ \ \ \ \ \ \ \ \ \ \ c++;
\end{align*}
\]

**Complexity:** \( O(n^2) \)

BUT, be careful with nests where the inner loop bounds depend on outer loop variables:

\[
\begin{align*}
\text{int } b &= n; \\
\text{while} (b>0)\{} \\
& \ \ \ b/=2; \\
& \ \ \ \text{for} (\text{int } i=0; i<b; i++) \\
& \ \ \ \ \ \ \ \ \ \ \ c++;
\end{align*}
\]

**Complexity:** \( n/2 + n/4 + \ldots + 2 + 1 = O(n) \)
Recursive Code

Draw the call tree, and assert the number of nodes in the tree and their individual complexity, as a function of n.
Draw the call tree, and assert the number of nodes in the tree and their individual complexity, as a function of $n$.

```java
public int divCo(int n){
    if(n<=1)
        return 1;
    else
        return 1 + divCo(n-1) + divCo(n-1);
}
```

How many recursive calls?

How much work per call?

What is the role of “return 1” and return 1+...”?

So what does this function count?

Complexity? $O(2^n)$
Polynomials.  $a_0 + a_1 n + \ldots + a_d n^d$ is $O(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant $d$

Logarithms.  $\log_a n$ is $O(\log_b n)$ for any constants $a, b > 0$.
  
  can avoid specifying the base
  
  for every $x > 0$, $\log n$ is $O(n^x)$.

Combinations. Merge sort, Heap sort $O(n \log n)$
A Survey of Common Running Times
Constant time: $O(1)$

A single line of code that involves “simple” expressions, e.g.:
- Arithmetical operations (+, -, *, /) for fixed size inputs
- assignments ($x = \text{simple expression}$)
- conditionals with simple sub-expressions
- function calls (excluding the time spent in the called function)
Logarithmic time

Example of a problem with $O(\log(n))$ bound:

binary search

How did we get that bound?

Repeated HALVING

How many times can I halve $n$ until I get to 1?

$log\ n$ times
Guessing game

I have a number between 0 and 63
How many (Y/N) questions do you need to find it?

What’s the number?

What (kind of) questions would you ask?
Guessing game

I have a number between 0 and 63
How many (Y/N) questions do you need to find it?

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>is it &gt;= 32</td>
<td>N</td>
</tr>
<tr>
<td>is it &gt;= 16</td>
<td>Y</td>
</tr>
<tr>
<td>is it &gt;= 24</td>
<td>N</td>
</tr>
<tr>
<td>is it &gt;= 20</td>
<td>N</td>
</tr>
<tr>
<td>is it &gt;= 18</td>
<td>Y</td>
</tr>
<tr>
<td>is it &gt;= 19</td>
<td>Y</td>
</tr>
</tbody>
</table>

What's the number?
19

Take N=0 and Y=1, what is 010011 ?

WHY?
When in each step of an algorithm we halve the size of the problem then it takes $\log_2 n$ steps to get to the base case.

We often use $\log(n)$ when we should use $\text{floor}(\log(n))$. That's OK since $\text{floor}(\log(n))$ is $\Theta(\log(n))$.

Similarly, if we divide a problem into $k$ equal parts the number of steps is $\log_k n$. For the purposes of big-O analysis it doesn't matter since $\log_a n$ is $O(\log_b n)$.  

log(n) and algorithms
Logarithms

definition:
\[ b^x = a \implies x = \log_b a, \text{ eg } 2^3 = 8, \log_2 8 = 3 \]

- \[ b^{\log_b a} = a \quad \log_b b = 1 \quad \log 1 = 0 \]
- \[ \log(x \cdot y) = \log x + \log y \text{ because } b^x \cdot b^y = b^{x+y} \]
- \[ \log(x/y) = \log x - \log y \]
- \[ \log x^a = a \log x \]
- \log x is a 1-to-1 monotonically (slow) growing function

\[ \log x = \log y \iff x = y \]

- \[ \log_a x = \log_b x / \log_b a \]
- \[ y^{\log x} = x^{\log y} \]
- \log x grows slower than any polynomial \( x^d \)
\[ \log_a x = \log_b x / \log_b a \]

\[ b^{\log_b x} = x = a^{\log_a x} = b^{(\log_b a)(\log_a x)} \]

\[ \log_b x = (\log_b a)(\log_a x) \]

\[ \log_a x = \log_b x / \log_b a \]

therefore \( \log_a x = O(\log_b x) \) for any \( a \) and \( b \)
\[ y^{\log x} = x^{\log y} \]

\[ x^{\log_b y} = \]

\[ y^{\log_y x \log_b y} = \]

\[ y^{(\log_b x / \log_b y) \log_b y} = \]

\[ y^{\log_b x} = \]
Linear Time: $O(n)$

**Linear time.** Running time is proportional to the size of the input.

**Computing the maximum.** Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

\[
\begin{aligned}
\text{max} & \leftarrow a_1 \\
\text{for } i = 2 \text{ to } n \{ & \\
\text{if } (a_i > \text{max}) & \\
\quad \text{max} & \leftarrow a_i
\}
\end{aligned}
\]

Also $\Theta(n)$?
Linear Time: $O(n)$

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into a single sorted list.

Claim. Merging two lists of size $n$ takes $O(n)$ time.
Polynomial evaluation. Given

\[ A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \quad (a_n \neq 0) \]

Evaluate \( A(x) \)

How not to do it:

\[ a_n \exp(x,n) + a_{n-1} \exp(x,n-1) + \ldots + a_1 x + a_0 \]

Why not?
How to do it: Horner's rule

\[a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 =
\]
\[(a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1) x + a_0 = \ldots =
\]
\[(\ldots(a_n x + a_{n-1}) x + a_{n-2}) x \ldots + a_1) x + a_0
\]

\[y = a[n]
\]
for \((i=n-1;i>=0;i--)
\]
\[y = y * x + a[i]
\]
Polynomial evaluation using Horner: complexity

Lower bound: $\Omega(n)$ because we need to access each $a[i]$ at least once

Upper bound: $O(n)$

Closed problem!

But what if $A(x) = x^n$
\[ A(x) = x^n \]

**Recurrence:**

\[ x^{2n} = x^n \cdot x^n \quad x^{2n+1} = x \cdot x^{2n} \]

```python
def pwr(x, n):
    if (n==0):
        return 1
    if odd(n):
        return x * pwr(x, n-1)
    else:
        a = pwr(x, n/2)
        return a * a
```

**Complexity?**
$O(n \log n)$ Time

Often arises in divide-and-conquer algorithms like mergesort.

```
mergesort(A):
    if len(A) <= 1 return A
    else return merge(mergesort(left half(A)),
                     mergesort(right half(A)))
```
Merge Sort - Divide

{7,3,2,9,1,6,4,5}

{7,3,2,9}  {1,6,4,5}

{7,3} {2,9}  {1,6} {4,5}

{7} {3} {2} {9} {1} {6} {4} {5}
Merge Sort - Merge
mergesort(A):
  if len(A) <= 1 return A
  else return merge(mergesort(left half(A)),
                    mergesort(right half(A)))
Quadratic time example. Enumerate all pairs of elements.

Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x_1 - x_2)^2 + (y_1 - y_2)^2
for i = 1 to n {
    for j = i+1 to n {
        d ← (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min ← d
    }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but ....
Example 1: Matrix multiplication

Tight?

Example 2: Set disjoint-ness. Given \( n \) sets \( S_1, \ldots, S_n \) each of which is a subset of \( 1, 2, \ldots, n \), is there some pair of these which are disjoint?

\( O(n^3) \) solution. For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set \( S_i \) {
    foreach other set \( S_j \) {
        foreach element \( p \) of \( S_i \) {
            determine whether \( p \) also belongs to \( S_j \)
        }
        if (no element of \( S_i \) belongs to \( S_j \))
            report that \( S_i \) and \( S_j \) are disjoint
    }
}
```

what do we need for this to be \( O(n^3) \)?
Polynomial Time: $O(n^k)$ Time

Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge?

$O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```
foreach subset S of k nodes {
    check whether S in an independent set
    if (S is an independent set)
        report S is an independent set
}
```

- Check whether $S$ is an independent set = $O(k^2)$.
- Number of $k$ element subsets = $\binom{n}{k} = \frac{n (n-1) (n-2) \cdots (n-k+1)}{k (k-1) (k-2) \cdots (2) (1)} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$.

poly-time for $k=17$, but not practical
Exponential Time

Independent set. Given a graph, what is the maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```plaintext
S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```

For some problems (e.g. TSP) we need to consider all $n!$ permutations of $n$ items.

The factorial function ($n!$) grows much faster than $2^n$
Questions

1. Is $2^n O(3^n)$?
2. Is $3^n O(2^n)$?
3. Is $2^n O(n!)$?
4. Is $n! O(2^n)$?
5. Is $\log_2 n O(\log_3 n)$?
6. Is $\log_3 n O(\log_2 n)$?
Some problems (such as matrix multiply) have a polynomial complexity solution: an $O(n^p)$ time algorithm solving them. ($p$ constant)

Some problems (such as Hanoi) take an exponential time to solve: $\Theta(p^n)$ ($p$ constant)

For some problems we only have an exponential solution, but we don't know if there exists a polynomial solution. Trial and error algorithms are the only ones we have so far to find an exact solution, and if we would always make the right guess, these algorithms would take polynomial time.

We call these problems NP (non deterministic polynomial) We will discuss NP later.
Some NP problems

**TSP**: Travelling Salesman
given cities \(c_1, c_2, \ldots, c_n\) and distances between all of these, find a minimal tour connecting all cities.

**SAT**: Satisfiability
given a boolean expression \(E\) with boolean variables \(x_1, x_2, \ldots, x_n\) determine a truth assignment to all \(x_i\) making \(E\) true.
Back tracking searches (walks) a state space, at each choice point it guesses a choice.

In a leaf (no further choices) if solution found OK, else go back to last choice point and pick another move.

NP is the class of problems for which we can check in polynomial time whether it is correct (certificates, later)
NP problems become intractable quickly

**TSP for 100 cities?**

How would you enumerate all possible tours? How many?

Coping with intractability:
- Approximation: Find a nearly optimal tour