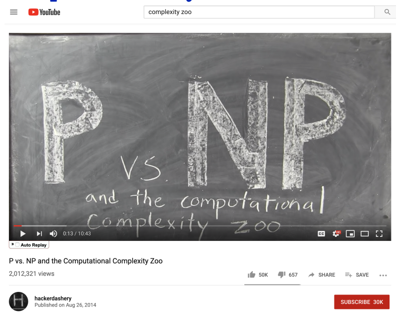
https://www.youtube.com/watch?v=YX40hbAHx3s


YouTube
complexity zoo
P vs. NP and the computational complexity zoo
0:13 / 10:42
Auto Replay
P vs. NP and the Computational Complexity Zoo
2,012,321 views
50K
657
SHARE
SAVE
hackerdashery
Published on Aug 26, 2014
SUBSCRIBE 30K

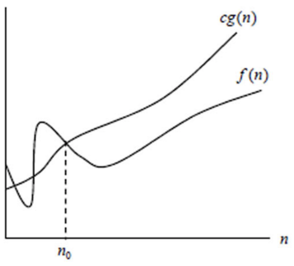
When considering a problem class, we want to know roughly how difficult the problem is. When considering an algorithm, we want to know roughly how fast the algorithm will run. These are questions that we can answer with complexity theory.

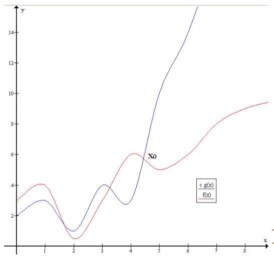
Binary: http://www.texample.net/tikz/examples/complement/

0.1 Big-O Notation

Definition 1 (Big-O). *For two functions $f(n)$ and $g(n)$, we say that $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that*

$$0 \le f(n) \le c\,g(n) \quad \text{for all} \quad n \ge n_0. \tag{0.1.1}$$





[1]

Figure 1: *Example of Big O notation: $f(x) \in O(g(x))$ as there exists $c > 0$ (e.g. $c = 1$) and x_0 (e.g. $x_0 = 5$) such that $f(x) \le cg(x)$ whenever $x \ge x_0$.*

We can also use Big-O to denote a set as

$$O(g(n)) := \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le c\,g(n), \text{ for all } n \ge n_0\}. \tag{0.1.2}$$

[1]Image borrowed from https://commons.wikimedia.org/wiki/File:Big-O-notation.png

Example 1: Consider $f(n) = 5n^2 + 10n + 7$ and $g(n) = n^2$. We want to show that $f(n) = O(g(n))$.

Let's try $c = 22$ and $n_0 = 1$. We need to show that **??** is satisfied.

Note first that we always have

1. $n^2 \leq n^2$ and therefore $5n^2 \leq 5n^2$

Note that if $n \geq 1$, then

2. $n \leq n^2$ and therefore $10n \leq 10n^2$
3. $1 \leq n^2$ and therefore $7 \leq 7n^2$

Since all inequalities 1,2, and 3 are valid for $n \geq 1$, by adding them, we obtain a new inequality that is also valid for $n \geq 1$, which is

$$5n^2 + 10n + 7 \leq 5n^2 + 10n^2 + 7n^2 \qquad \text{for all } n \geq 1, \tag{0.1.3}$$

$$\Rightarrow \quad 5n^2 + 10n + 7 \leq 22n^2 \qquad \text{for all } n \geq 1. \tag{0.1.4}$$

Hence, we have shown that **??** holds for $c = 22$ and $n_0 = 1$. Hence $f(n) = O(g(n))$.

Correct uses:

- $2^n + n^5 + \sin(n) = O(2^n)$
- $2^n = O(n!)$
- $n! + 2^n + 5n = O(n!)$
- $n^2 + n = O(n^3)$
- $n^2 + n = O(n^2)$
- $\log(n) = O(n)$
- $10\log(n) + 5 = O(n)$

Notice that not all examples above give a tight bound on the asymptotic growth. For instance, $n^2 + n = O(n^3)$ is true, but a tighter bound is $n^2 + n = O(n^2)$.

In particular, the goal of big O notation is to give an upper bound on the asymptotic growth of a function. But we would prefer to give a strong upper bound as opposed to a weak upper bound. For instance, if you order a package online, you will typically be given a bound on the latest date that it will arrive. For example, if it will arrive within a week, you might be guaranteed that it will arrive by next Tuesday. This sounds like a reasonable bound. But if instead, they tell you it will arrive before 1 year from today, this may not be as useful information. In the case of big O notation, we would like to give a least upper bound that most simply describes the growth behavior of the function.

In that example, $n^2 + n = O(n^2)$, this literally means that there is some number c and some value n_0 that $n^2 + n \leq cn^2$ for all $n \geq n_0$, that is, for all values of n_0 larger than n, the function cn^2 dominates $n^2 + n$.

For example, a valid choice is $c = 2$ and $n_0 = 1$. Then it is true that $n^2 + n \leq 2n^2$ for all $n \geq 1$.

But it is also true that $n^2 + n = O(n^3)$. For example, a valid choice is again $c = 2$ and $n_0 = 1$, then

$n^2 + n \leq 2n^3$ for all $n \geq 1$.

In this example, $O(n^3)$ is the case where the internet tells you the package will arrive before 1 year from today. The bound is true, but it is not as useful information as we would like to have. Let's compare these upper bounds. Let $f(n) = n^2 + n$, $g(n) = 2n^2$, $h(n) = 2n^3$.

Then we have

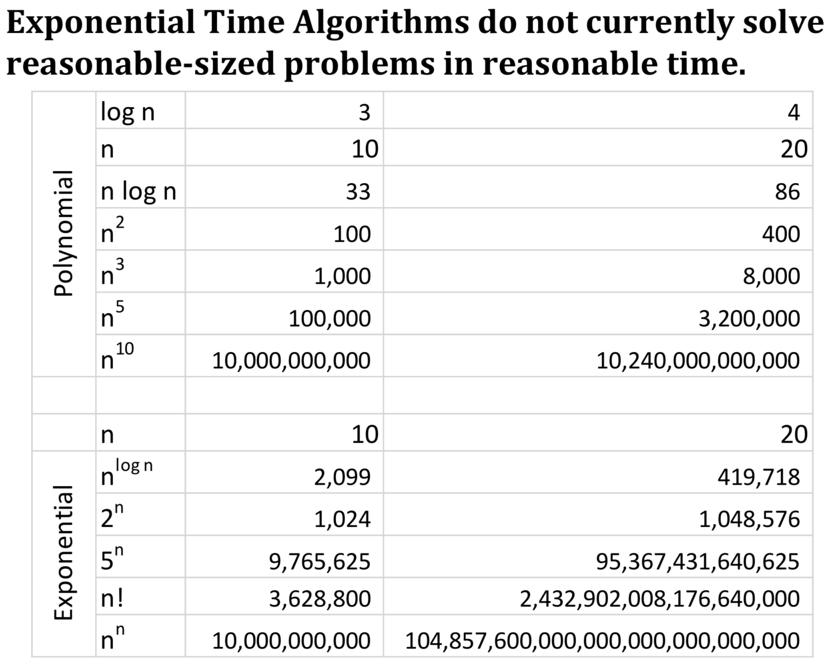
```
                n = 10 .        n = 100 .       n = 1000 .       n = .  10000

f(n)            110,          10100,         1001000,              100010000

g(n)           200,          20000,        2000000,                200000000

h(n) .         2000,        2000000,      2000000000,      2000000000000
```

So, here we see that $g(n)$ and $h(n)$ are both upper bounds on $f(n)$, but the nice part about $g(n)$ is that is growing at a similar rate to $f(n)$. In particular, it is always within a factor of 2 of $f(n)$.

Alternatively, the bound h(n) is true, but it grows so much faster than $f(n)$ that is doesn't give a good idea of the asymptotic growth of $f(n)$.

Some common classes of functions:

$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n^c)$ (for $c > 1$)	Polynomial
$O(c^n)$ (for $c > 1$	Exponential



Exponential Time Algorithms do not currently solve reasonable-sized problems in reasonable time.

Polynomial	log n	3	4
	n	10	20
	n log n	33	86
	n^2	100	400
	n^3	1,000	8,000
	n^5	100,000	3,200,000
	n^{10}	10,000,000,000	10,240,000,000,000
	n	10	20
Exponential	$n^{\log n}$	2,099	419,718
	2^n	1,024	1,048,576
	5^n	9,765,625	95,367,431,640,625
	n!	3,628,800	2,432,902,008,176,640,000
	n^n	10,000,000,000	104,857,600,000,000,000,000,000,000

0.2 Algorithms - Example with Bubble Sort

The following definition comes from Merriam-Webster's dictionary.

Definition 2. *An algorithm is a procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end.*

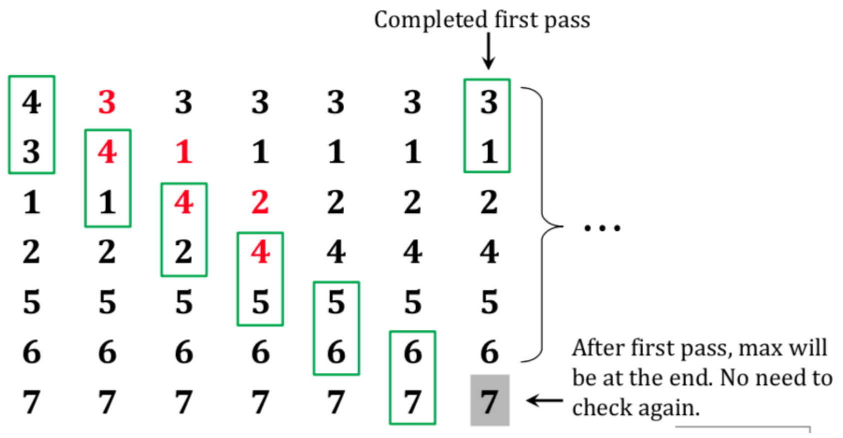
0.2.1 Sorting

https://en.wikipedia.org/wiki/Bubble_sort

Bubble sort algorithm:.....

Bubble sort algorithm

- Compare two neighboring objects and swap if they are in the wrong order.


Completed first pass
…
After first pass, max will be at the end. No need to check again.

The algorithm will terminate when a pass is completed with no swaps.

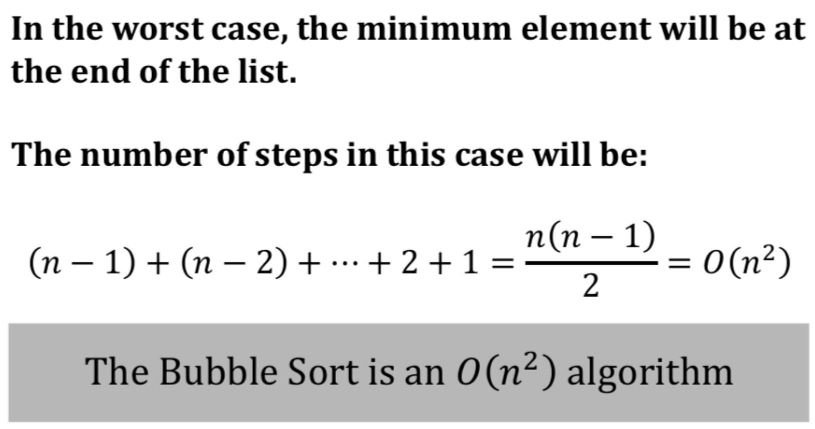
How many steps did it take to sort the numbers in this example?

15 steps

This is around $2n$ ($n = 7$). Does this mean it is $O(n)$?

NO!

The worst case must be examined.



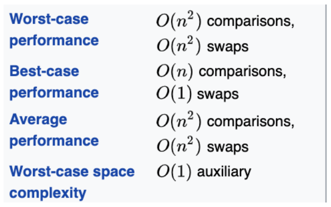
In the worst case, the minimum element will be at the end of the list.

The number of steps in this case will be:

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=O(n^2)$$

The Bubble Sort is an $O(n^2)$ algorithm

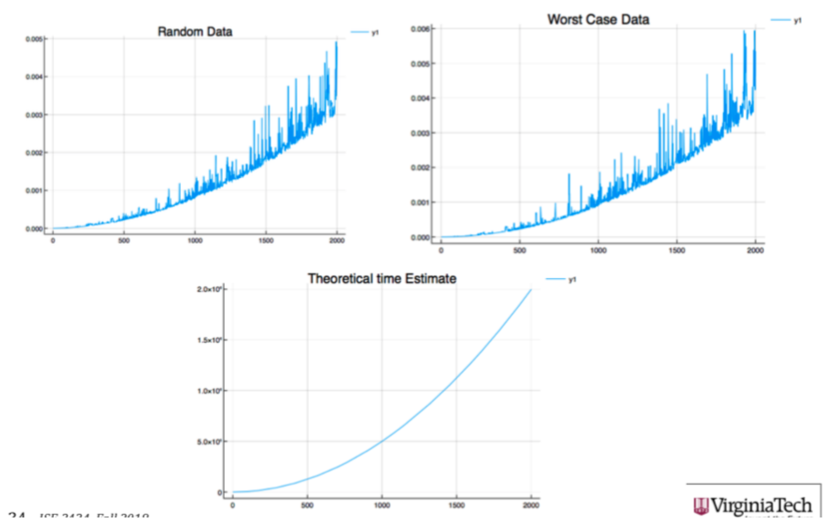
We can also consider the best case and average case complexity:



Worst-case performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Worst-case space complexity	$O(1)$ auxiliary

These can be verified experimentally as seen in the following plot. The random case grows quadratically just as the worst case does.

Time elapsed in computer for bubble sort



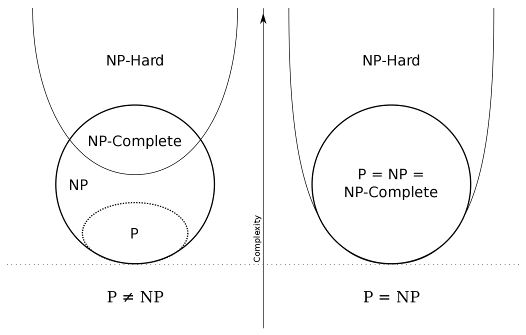
0.3 Complexity Classes

In this subsection we will discuss the complexity classes P, NP, NP-Complete, and NP-Hard. These classes help measure how difficult a problem is. **Informally,** these classes can be thought of as

- P - the class of efficiently solvable problems
- NP - the class of efficiently checkable problems

- NP-Hard - the class of problems that can solve any problem in NP
- NP-Complete - the class of problems that are in both NP and are NP-Hard.

It is not known if P is the same as NP, but it is conjectured that these are very different classes. This would mean that the NP-Hard problems (and NP-Complete problems) are necessarily much more difficult than the problems in P.



[2]

Figure 2: *Complexity class possibilities. Most academics agree that the case $P \neq NP$ is more likely.*

We will now discuss these classes more formally.

0.3.1 P

P is the class of polynomially solvable problems. P contains tall problems for which there exists an algorithm that solves the problem in a run time bounded by a polynomial. That is, $O(n^c)$ for some constant c.

Example 2: The minimum size spanning tree problem is in P. It can be solved, for instance, by Prim's algorithm, which by runs in time $O(m \log n)$, where m is the number of edges in the graph and n is the number of nodes in the graph.

Example 3: Linear programming is in P. It can be solved by interior point methods in $O(n^{3.5}\phi)$ where ϕ represents the number of binary bits that are required to encode the problem. These bits describe the matrix A, and vectors c and b that define the linear program.

[2]Figure from https://en.wikipedia.org/wiki/NP_(complexity)

0.3.2 NP

NP is the class of nondeterministic polynomial problems. NP contains all problems in which membership can be verified in polynomial time from a certificate.

Thus, to show that a problem is in NP, you must do the following:

1. Describe a format for a certificate to the problem.
2. Show that given such a certificate, it is easy to verify the solution to the problem.

Example 4: Integer Linear Programming is in NP. More explicitly, the feasibility question of

"Does there exists an integer vector x that satisfies $Ax \leq b$"

is in NP.

Although it turns out to be difficult to find such an x or even prove that one exists, this problem is in NP for the following reason: if you are given a particular x and someone claims to you that it is a feasible solution to the problem, then you can easily check if they are correct. In this case, the vector x that you were given is called a *certificate*.

Note that it is easy to verify if x is a solution to the problem because you just have to

1. Check if x is integer.
2. Use matrix multiplication to check if $Ax \leq b$ holds.

0.3.3 NP-Hard

The class of problems that are called *NP-Hard* are those that can be used to solve any other problem in the NP class. That is, problem A is NP-Hard provided that for any problem B in NP there is a transformation of problem B that preserves the size of the problem, up to a polynomial factor, into a new problem that problem A can be used to solve.

Here we think of “if problem A could be solved efficiently, then all problems in NP could be solved efficiently".

More specifically, we assume that we have an oracle for problem A that runs in polynomial time. An oracle is an algorithm that for the problem that returns the solution of the problem in a time polynomial in the input. This oracle can be thought of as a magic computer that gives us the answer to the problem. Thus, we say that problem A is NP-Complete provided that given an oracle for problem A, one can solve any other problem B in NP in polynomial time.

Note: These problems are not necessarily in NP.

0.3.4 NP-Complete

The class of problems that are call *NP-Complete* are those which are in NP and also NP-Hard.

We know of many problems that are NP-Complete. For example, binary integer programming feasibility is NP-Complete. One can show that another problem is NP-complete by

1. showing that it can be used to solve binary integer programming feasibility,
2. showing that the problem is in NP.

The first problem proven to be NP-Complete is called *3-SAT* []. 3-SAT is a special case of the *satisfiability problem*. In a satisfiability problem, we have variables $X_1, \dots, X_n$ and we want to assign them values as either `true` or `false` . The problem is described with *AND* operations, denoted as $\wedge$, with *OR* operations, denoted as $\vee$, and with *NOT* operations, denoted as $\neg$. The *AND* operation $X_1 \wedge X_2$ returns `true` if BOTH X_1 and X_2 are true. The *OR* operation $X_1 \vee X_2$ returns `true` if AT LEAST ONE OF X_1 and X_2 are true. Lastly, the *NOT* operation $\neg X_1$ returns there opposite of the value of X_1.

These can be described in the following table

$$\texttt{true} \wedge \texttt{true} = \texttt{true} \tag{0.3.1}$$
$$\texttt{true} \wedge \texttt{false} = \texttt{false} \tag{0.3.2}$$
$$\texttt{false} \wedge \texttt{false} = \texttt{false} \tag{0.3.3}$$
$$\texttt{false} \wedge \texttt{true} = \texttt{false} \tag{0.3.4}$$

$$\texttt{true} \vee \texttt{true} = \texttt{true} \tag{0.3.5}$$
$$\texttt{true} \vee \texttt{false} = \texttt{true} \tag{0.3.6}$$
$$\texttt{false} \vee \texttt{false} = \texttt{false} \tag{0.3.7}$$
$$\texttt{false} \vee \texttt{true} = \texttt{true} \tag{0.3.8}$$

$$\neg\, \texttt{true} = \texttt{false} \tag{0.3.9}$$
$$\neg\, \texttt{false} = \texttt{true} \tag{0.3.10}$$

For example, A *logical expression* is a sequence of logical operations on variables $X_1, \dots, X_n$, such that

$$(X_1 \wedge \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \vee (X_1 \wedge X_2 \wedge X_3). \tag{0.3.11}$$

A *clause* is a logical expression that only contains the operations $\vee$ and $\neg$ and is not nested (with parentheses), such as

$$X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4. \tag{0.3.12}$$

A fundamental result about logical expressions is that they can always be reduced to a sequence of clauses that are joined by $\wedge$ operations, such as

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_2 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5). \tag{0.3.13}$$

The satisfiability problem takes as input a logical expression in this format and asks if there is an assignment of `true` or `false` to each variable X_i that makes the expression `true` . The 3-SAT problem is a special case where the clauses have only three variables in them.

3-SAT:
NP-Complete
Given a logical expression in n variables where each clause has only 3 variables, decide if there is an assignment to the variables that makes the expression `true` .

Binary Integer Programming:
NP-Complete
Binary Integer Programming can easily be shown to be in NP, since verifying solutions to BIP can be done by checking a linear system of inequalities.

Furthermore, it can be shown to be NP-Complete since it can be used to solve 3-SAT. That is, given an oracle for BIP, since 3-SAT can be modeled as a BIP, the 3-SAT could be solved in oracle-polynomial time.

0.4 Relevant Terminology

We will discuss the following concepts:

- Feasible solutions
- Optimal solutions
- Approximate solutions
- Heuristics
- Exact Algorithms
- Approximation Algorithms
- Complexity class relations

0.5 Matching Problem

Definition 3. *Given a graph $G = (V, E)$, a* matching *is a subset $E' \subseteq E$ such that no vertex $v \in V$ is contained in more than one edge in E'.*
A perfect matching *is a matching were every vertex is connected to an edges in E'.*
A maximal matching *is a matching E' such that there is no matching E'' that strictly contains it.*

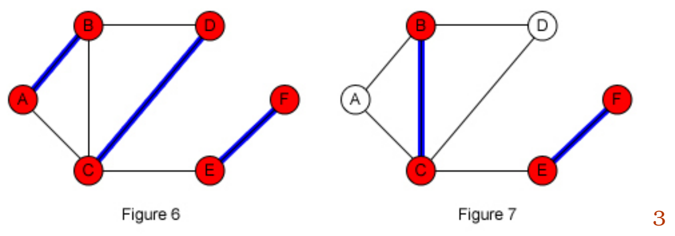
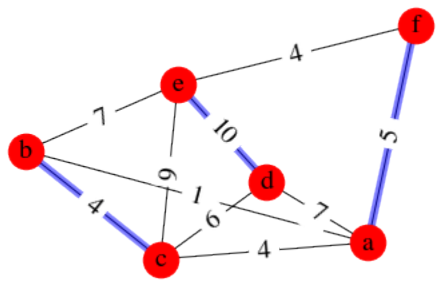


Figure 3: *Two possible matchings. On the left, we have a perfect matchings (all nodes are matched). On the right, a feasible matching, but not a perfect matching since not all nodes are matched.*

Definition 4. *Maximum Weight Matching Given a graph $G = (V, E)$, with associated weights $w_e \geq 0$ for all $e \in E$, a* maximum weight matching *is a matching that maximizes the sum of the weights in the matching.*



0.5.1 Greedy Algorithm for Maximal Matching

The greedy algorithm iteratively adds the edge with largest weight that is feasible to add.

Greedy Algorithm for Maximal Matching:
Complexity: $O(|E|\log(|V|))$

1. Begin with an empty graph $(M = \emptyset)$
2. Label the edges in the graph such that $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_m)$
3. For $i = 1, \ldots, m$
   If $M \cup \{e_i\}$ is a valid matching (i.e., no vertex is incident with two edges), then set $M \leftarrow M \cup \{e_i\}$ (i.e., add edge e_i to the graph M)
4. Return M

[3]Figure from http://www.suhendry.net/blog/?p=1647.

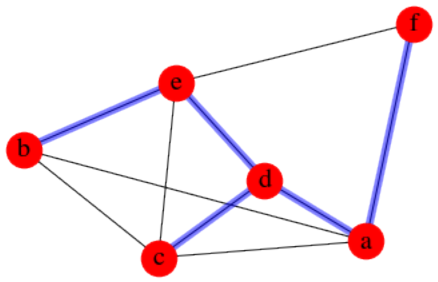
Theorem 5 ([**?**]). *The greedy algorithm finds a 2-approximation of the maximum weighted matching problem. That is,* $w(M_{greedy}) \geq \frac{1}{2}w(M^*)$.

0.5.2 Other algorithms to look at

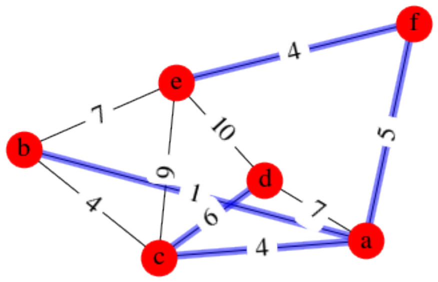
1. Improved Algorithm [**?**]
2. Blossom Algorithm https://en.wikipedia.org/wiki/Blossom_algorithm

0.6 Minimum Spanning Tree

Definition 6. *Given a graph* $G = (V, E)$, *a* spanning tree *connected, acyclic subgraph* T *that contains every node in* V.



Definition 7. *Given a graph* $G = (V, E)$, *with associated weights* $w_e \geq 0$ *for all* $e \in E$, *a* maximum weight spanning tree *is a spanning tree maximizes the sum of the edge weights.*



Lemma 8. *Let* G *be a connected graph with* n *vertices.*

1. T *is a spanning tree of* G *if and only if* T *has* $n - 1$ *edges and is connected.*
2. *Any subgraph* S *of* G *with more than* $n - 1$ *edges contains a cycle.*

See Section **??** for integer programming formulations of this problem.

0.6.1 Kruskal's algorithm

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Kruskal - for Minimum Spanning tree:
Complexity: $O(|E|\log(|V|)$

1. Begin with an empty tree $(T = \emptyset)$
2. Label the edges in the graph such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
3. For $i = 1, \dots, m$
   If $T \cup \{e_i\}$ is acyclic, then set $T \leftarrow T \cup \{e_i\}$
4. Return T

0.6.2 Prim's Algorithm

https://en.wikipedia.org/wiki/Prim%27s_algorithm
http://www.texample.net/tikz/examples/prims-algorithm/

0.7 Traveling Salesman Problem

https://www.youtube.com/watch?v=CPetTODX-FA https://www.youtube.com/watch?v=R_IfyticWKQ https://www.youtube.com/watch?v=2jncD54ryGs

See Section **??** for integer programming formulations of this problem. Also, hill climbing algorithms for this problem such as 2-Opt, simulated annealing, and tabu search will be discussed in Section **??**.

0.7.1 Nearest Neighbor - Construction Heuristic

https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm Starting from any node, add the edge to the next closest node. Continue this process.

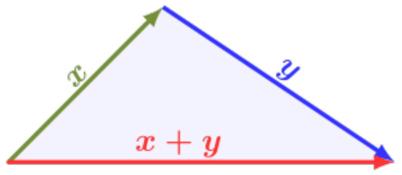
Nearest Neighbor:
Complexity: $O(n^2)$

1. Start from any node (lets call this node 1) and label this as your current node.
2. Pick the next current node as the one that is closest to the current node that has not yet been visited.
3. Repeat step 2 until all nodes are in the tour.

0.7.2 Double Spanning Tree - 2-Apx

Graphs with nice properties are often easier to handle and typically graphs found in the real world have some nice properties. The *triangle inequality* comes from the idea of a triangle that the sum of the lengths of two sides always are longer than the length of the third side. That is for side lengths x, y, z, we have

$$z \leq x + y$$



[4]

Definition 9. *A complete, weighted graph G (i.e., a graph that has all possible edges and a weight assigned to each edge) satisfies the* triangle inequality *provided that for ever triple of vertices a, b, c and edges e_{ab}, e_{bc}, e_{ac}, we have that*

$$w(e_{ab}) + w(e_{bc}) \geq w(e_{ac}).$$

Algorithm 1 Double Spanning Tree

```
Input: A graph $G = (V, E)$ that satisfies the triangle inequality
Output: A tour that is a 2-Apx of the optimal solution
1: Compute a minimum spanning tree $T$ of $G$.
2: Double each edge in the minimum spanning tree (i.e., if edge $e_{ab}$ is in $T$, add edge $e_{ba}$.
3: Compute an Eulerian Tour using these edges.
4: Return tour that visits vertices in the order the Eulerian Tour visits them, but without
   repeating any vertices.
```

Let S be the resulting tour and let S^* be an optimal tour. Since the resulting tour is feasible, it will satisfy

$$w(S^*) \leq w(S).$$

But we also know that the weight of a minimum spanning tree T is less than that of the optimal tour, hence

$$w(T) \leq w(S^*).$$

Lastly, due to the triangle inequality we know that

$$w(S) \leq 2w(T),$$

since replacing any edge in the Eulerian tour with a more direct edge only reduces the total weight.

Putting this together, we have

$$w(S) \leq 2w(T) \leq 2w(S^*)$$

and hence, S is a 2-approximation of the optimal solution.

[4]Figure borrowed from https://en.wikipedia.org/wiki/Triangle_inequality#/media/File:Vector-triangle-inequality.svg

0.7.3 Christofides - Approximation Algorithm - $(3/2)$-Apx

If we combine algorithms for minimum spanning tree and matching, we can find a better approximation algorithm. This is given by Christofides. Again, this is in the case where the graph satisfies the triangle inequality. See `https://en.wikipedia.org/wiki/Christofides_algorithm` or `https://resources.mpi-inf.mpg.de/conferences/adfocs-15/material/Ola-Lect1.pdf` for more detail.