Combinatorial Problems

A problem like Independent Set, Stable Marriage, etc. can be thought of as families of instances:

- An instance of Independent Set is specified by a particular graph $G$.
- An instance of Stable Matching is given by the preference lists.

These instances often will have a natural way of encoding them into the computer:

- A graph with $N$ nodes might be specified by an $N \times N$ matrix or a list of edges.
- The preference lists could be encoded as an array of lists, or a matrix.
Size of the instance is the space used to represent it.

Usually use \( n \) to represent this size.

Generally, “larger” instances are more difficult than smaller ones.

Can often break the problem down into:

- **Search space**: the set of feasible solutions (every independent set or perfect matching)
- **Objective function**: a way of measuring how good the solution is (size of independent set, stability of perfect matching)
Central Dogma of Computer Science:

Difficulty is not necessarily proportional to size of the search space.
Efficient Algorithms

Definition (Efficiency)
An algorithm is efficient if its worst-case runtime is bounded by a polynomial function of the input size.

- There is a polynomial $p(n)$ such that for every instance of size $n$, the algorithm solves the instance in fewer than $p(n)$ steps.

Generally, the problems we consider will have huge search spaces.

How many perfect matchings are there in a stable matching problem with $n$ men and $n$ women?

Any polynomial is much smaller than the search space.
An algorithm is efficient if its worst-case runtime is bounded by a polynomial function of the input size.

- There is a polynomial $p(n)$ such that for every instance of size $n$, the algorithm solves the instance in fewer than $p(n)$ steps.

Generally, the problems we consider will have huge search spaces.

How many perfect matchings are there in a stable matching problem with $n$ men and $n$ women? $n!$

Any polynomial is much smaller than the search space.
This definition of efficiently is not perfect:

1. There are non-polynomial algorithms that are usually the best choice in practice (e.g. simplex method for linear programming).

2. There are polynomial time algorithms that are almost never used in practice (e.g. ellipsoid algorithm for linear programming).

3. An algorithm that takes \( n^{100} \) steps would be “efficient” by this definition, but horrible in practice.

4. An algorithm that takes \( n^{1+0.002 \log n} \) is probably useful in practice, but it is not “efficient”. 
But it has a lot of benefits:

1. It’s concrete and falsifiable — avoids “vague” arguments about which algorithm is better.

2. Average performance is hard to define.

3. The exceptions are apparently fewer than the cases where polynomial time corresponds to useful algorithms in practice.

4. There is normally a huge difference between polynomial time and other natural runtimes

   (If you can do 1 million steps per second and $n = 1,000,000$, then a $n^2$ algorithm would take 12 days, but a $1.5^n$ algorithm would take far more than $10^{25}$ years)
A running time of $n^2 + 4n + 2$ is usually too detailed. Rather, we’re interested in how the runtime grows as the problem size grows.

**Definition (O)**

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

$$T(n) \leq cf(n) \text{ for all } n \geq n_0$$

What does this mean?

- for all large enough instances
- the running time is bounded by a constant multiple of $f(n)$
Asymptotic Lower Bounds

\(O(\cdot)\) talks about the \textit{longest} possible time an algorithm could take. \(\Omega(\cdot)\) talks about the \textit{shortest} possible time.

\textbf{Definition (\(\Omega\))}

\(T(n)\) is \(\Omega(f(n))\) if there are constants \(\epsilon > 0\) and \(n_0 \geq 0\) so that:

\[ T(n) \geq \epsilon f(n) \quad \text{for all } n \geq n_0 \]
Definition (Θ)

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

If we know that \( T(n) \) is \( \Theta(f(n)) \) then \( f(n) \) is the “right” asymptotic running time: it will run faster than \( O(f(n)) \) on all instances and some instances might take that long.
Asymptotic Limit

**Theorem**

If \( \lim_{n \to \infty} \frac{T(n)}{g(n)} \) equals some \( c > 0 \), then \( T(n) = \Theta(f(n)) \).

There is an \( n_0 \) such that \( \frac{c}{2} \leq \frac{T(n)}{g(n)} \leq 2c \) for all \( n \geq n_0 \).

Therefore, \( T(n) \leq 2cg(n) \) for \( n \geq n_0 \) \( \Rightarrow \) \( T(n) = O(g(n)) \).

Also, \( T(n) \geq \frac{c}{2}g(n) \) for \( n \geq n_0 \) \( \Rightarrow \) \( T(n) = \Omega(g(n)) \).
Linear Time

Linear time usually means you look at each element a constant number of times.

- Finding the maximum element in a list:

  ```
  max = a[1]
  for i = 2 to n:
      if a[i] > max then
          set max = a[i]
  Endfor
  ```

  This does a constant amount of work per element in array a.

- Merging sorted lists.
$O(n \log n)$ time common because of sorting (often the slowest step in an algorithm).

Where does the $O(n \log n)$ come from?
$O(n \log n)$ time common because of sorting (often the slowest step in an algorithm).

Where does the $O(n \log n)$ come from?

$T(n) = 2T(n/2) + n$
Given a set of points, what is the smallest distance between them: \( \binom{n}{2} \) pairs.
Given $n$ sets $S_1, S_2, \ldots, S_n$ that are subsets of $\{1, \ldots, n\}$, is there some pair of sets that is disjoint?
Larger polynomials arise from exploring smaller search spaces exhaustively:

**Independent Set of Size $k$**

Given a graph with $n$ nodes, find an independent set of size $k$ or report none exists.

For every subset $S$ of $k$ nodes

If $S$ is an independent set then

return $S$

Endfor

return Failure

How many subsets of $k$ nodes are there?
Exponential Time

What if we didn't limit ourselves to independent sets of size $k$, and instead want to find the largest independent set?

**Brute force** algorithms search through all possibilities.

How many subsets of nodes are there in an $n$-node graph?

What’s the runtime of the brute force search search for a largest independent set?
Exponential Time

What if we didn’t limit ourselves to independent sets of size $k$, and instead want to find the largest independent set?

**Brute force** algorithms search through all possibilities.

How many subsets of nodes are there in an $n$-node graph? $2^n$

What’s the runtime of the brute force search search for a largest independent set?
What if we didn't limit ourselves to independent sets of size $k$, and instead want to find the largest independent set?

**Brute force** algorithms search through all possibilities.

How many subsets of nodes are there in an $n$-node graph? $2^n$

What's the runtime of the brute force search search for a largest independent set? $O(n^2 2^n)$
Sublinear time means we don’t even look at every input.

Since it takes $n$ time just to read the input, we have to work in a model where we count how many queries to the data we make.

Sublinear usually means we don’t have too look at every element.

Example?
Sublinear time means we don’t even look at every input.

Since it takes $n$ time just to read the input, we have to work in a model where we count how many queries to the data we make.

Sublinear usually means we don’t have too look at every element.

Example? Binary search — $O(\log n)$