Lecture 3

Overview, greedy algorithms, and graph traversal

Chinmay Nirkhe | CSE 421 Spring 2025

Previously in CSE 421…

Algorithmic complexity

Measuring algorithmic efficiency

The RAM model

- RAM Model = "Random Access Machine" Model
- Each simple operation (arithmetic, evaluating if loop criteria, call, increment counter, etc.) takes one time step
- Accessing any one arithmetic number in memory takes one time step
- Measuring algorithm efficiency
  - Let input be $(x_1, \ldots, x_n)$ with each x_i representing one arithmetic number
  - Runtime of algorithm is the number of "simple operations" taken to compute algorithm in RAM model.

Today

Complexity analysis

- Input $(x_1, \ldots, x_n)$ of length n.
- Multiple measures of complexity.
    - Worst-case: **maximum** # of steps taken on *any* input of length n
    - Best-case: **minimum** # of steps taken on *any* input of length n
    - Average-case: **average** # of steps taken over *all* input of length n

Complexity analysis

- The complexity of an alg. is a function $T(n)$ for each input size $n \in \mathbb{N}$.
- i.e. $T_{\text{worst}}(n)$ or $T_{\text{avg}}(n)$ could be two different functions.
- $T : \mathbb{N} \to \mathbb{N}$
- We are interested in understanding the overall behavior/shape of T, not the exact function.
- Sometimes there is more than one size parameter. $T(n, m)$ for a n vertex and m edge graph.

Polynomial time

A notion of efficiency

- A function $T(n)$ is **polynomial time** if $T(n) \leq cn^k + d$ for some constants $c, k, d > 0$.
  - Let k be the minimal such value. This is the degree of the *dominating* polynomial.
  - Polynomial time is known as "efficient" in theoretical CS.

Polynomial time

A notion of efficiency

- A function $T(n)$ is **polynomial time** if $T(n) \leq cn^k + d$.
- Why **polynomial time**?
    - Scaling the instance by a constant factor so does the runtime.
    - **Church-Turing thesis**: Any function computable in polynomial time by a physically realizable model of computation can also be computed in polynomial time a *different* physically realizable model.
        - I.e. polynomial-time is a notion independent of model of computation.
        - Ideal for theoretical study of what problems are efficient and which are not.
    - Problem size grows by constant, then running time also grows by constant.
    - If $T(n) = cn^k + d$ then $T(2n) = c(2n)^k + d \leq 2^k(cn^k + d) = 2^kT(n)$.
    - Typically, polynomials for common algorithms are small polynomials cn, cn^2, cn^3, cn^4. Rarely anything higher.

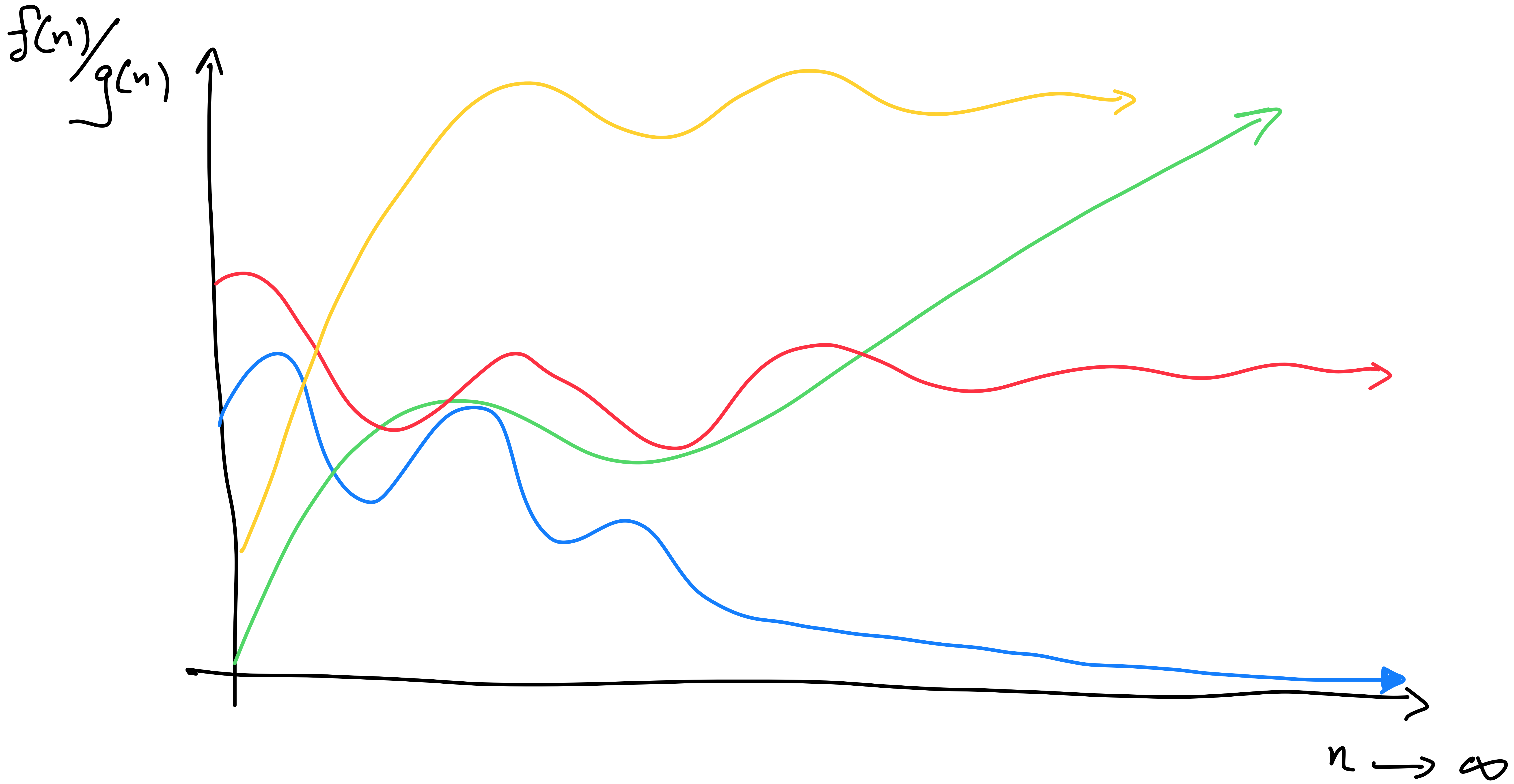
Big-O notation

Let $T, g : \mathbb{N} \to \mathbb{N}$. Then

- $T(n)$ is $O(g(n))$ if $\exists\ c, n_0 > 0$ such that $T(n) \leq cg(n)$ when $n \geq n_0$.
- $T(n)$ is $o(g(n))$ if $\displaystyle\lim_{n\to\infty} \frac{T(n)}{g(n)} = 0$.
- $T(n)$ is $\Omega(g(n))$ if $\exists\ \epsilon, n_0 > 0$ such that $T(n) \geq \epsilon g(n)$ when $n \geq n_0$.
- $T(n)$ is $\Theta(g(n))$ if $T(n)$ is $O(g(n))$ **and** $T(n)$ is $\Omega(g(n))$.

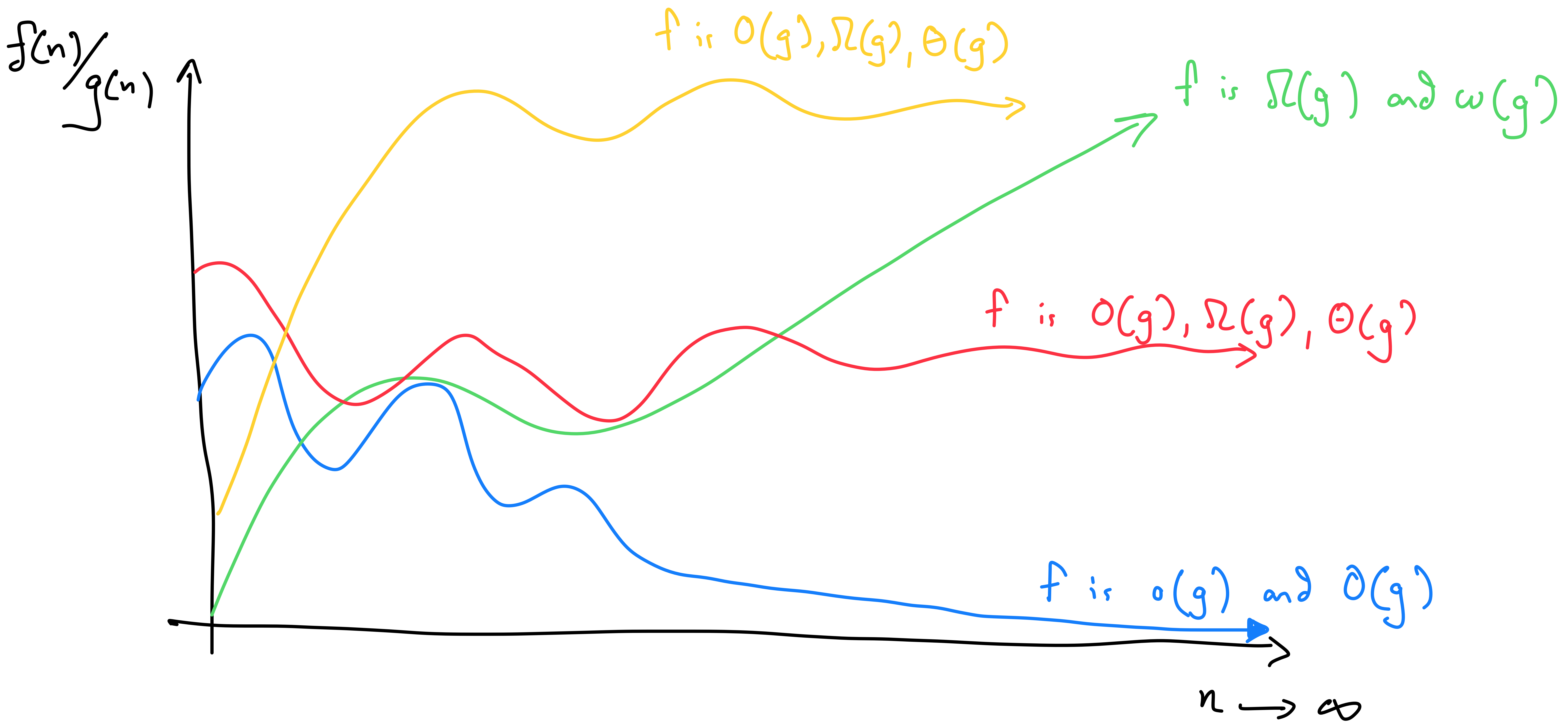
Big-O notation

Cartoon



Big-O notation

Cartoon



Measuring algorithmic efficiency

The RAM model

- RAM Model = “Random Access Machine” Model
- Each simple operation (arithmetic, evaluating if loop criteria, call, increment counter, etc.) takes one time step
- Accessing any bit of memory takes one time step

Measuring algorithmic efficiency

The RAM model, Examples

- Sorting a list of integers $L = (x_1, \ldots, x_n)$
  - You probably know that sorting can be solved in $\Theta(n \log n)$ time by algorithms such as merge sort.
  - This is measuring the number of comparisons $x_i < x_j$ that we are making. RAM model makes this rigorous.
- All-pairs shortest path problem: Given a weighted graph $G = (V, E)$ output $d_{uv} = \min_{p: u \rightsquigarrow v} \sum_{(a,b) \in p} w_{ab}$ for every pair of vertices $u, v \in V$.
  - Floyd-Warshall alg. Makes $O(n^3)$ arithmetic comparisons where $n = |V|, m = |E|$.
  - Requires adjacency matrix access to the graph. Meaning, unit cost to compute w_{ab} for any $a, b \in V$.

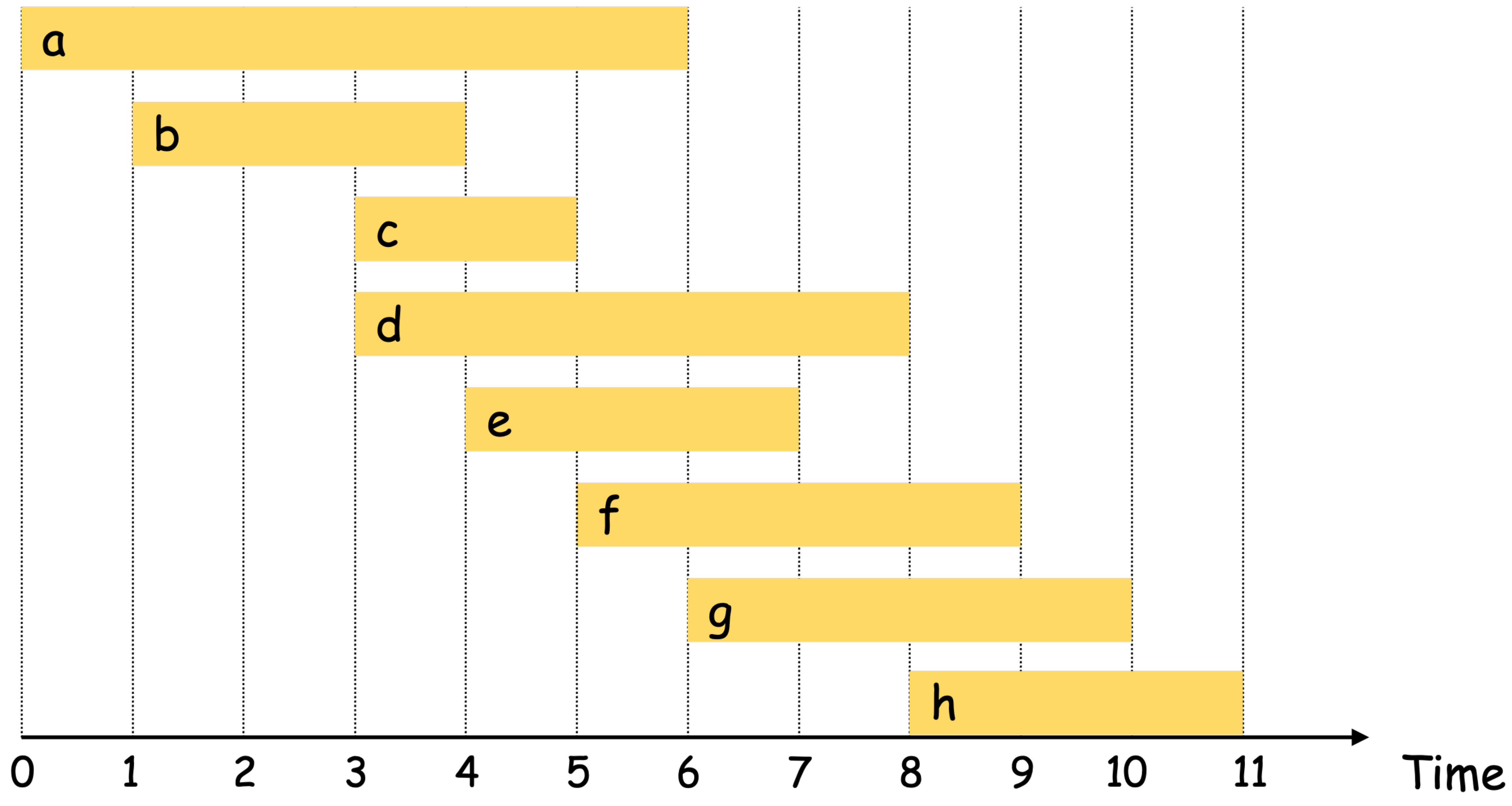
An introduction to algorithms

An introduction to algorithms

- Goal is to understand *how* to analyze and *design* algorithms
  - To understand how small changes have big effects on outcomes
  - Build a repertoire of techniques for designing algorithms
  - Identifying when to use which family of algorithms
- Course is structured by teaching various families of algorithms
  - Section and problem sets will cover example instantiations pertinent to that week
  - Midterms and finals will have problems but won't say which family of algorithms to use

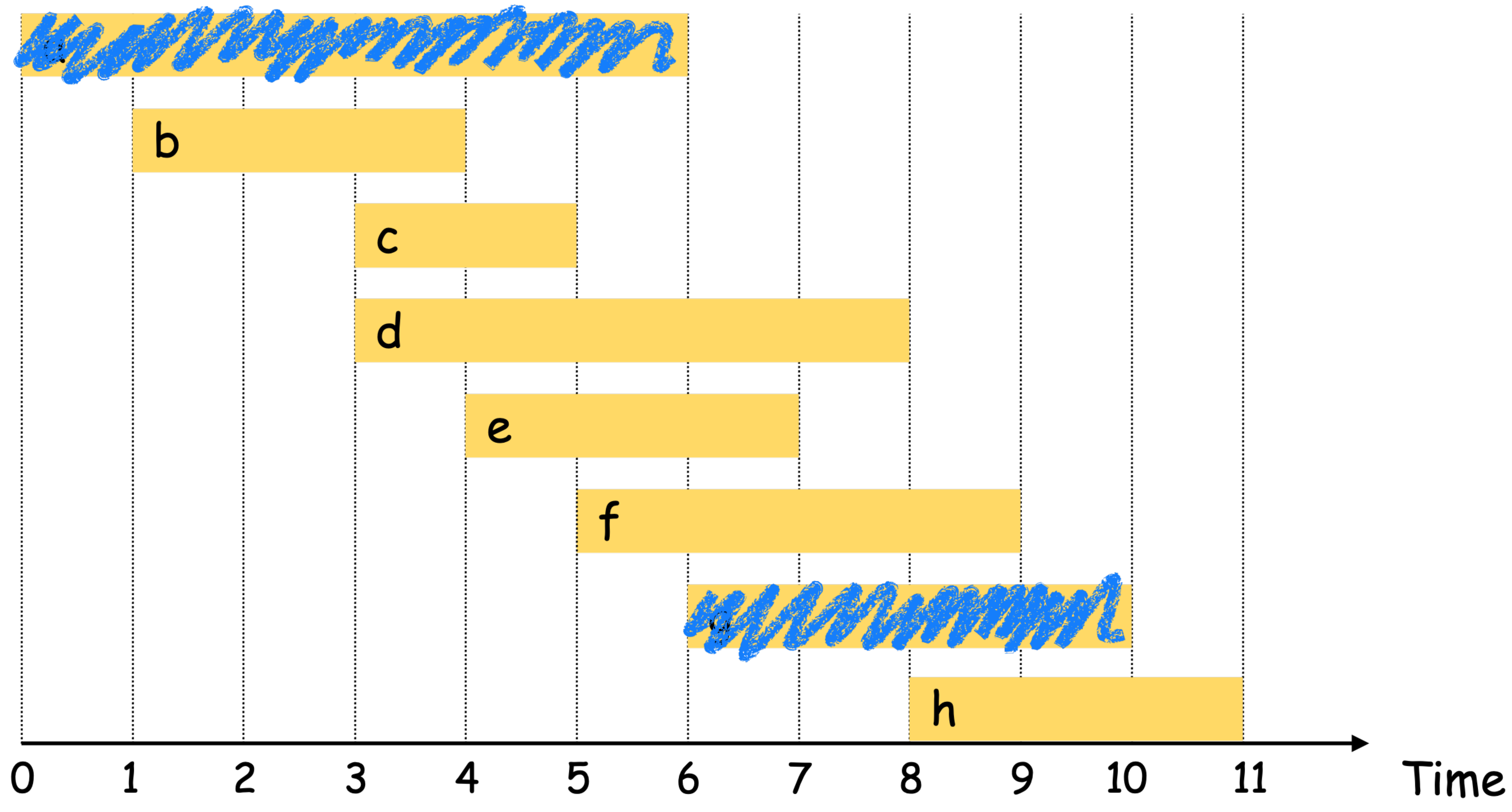
Interval scheduling

- **Input:** start and end times (s_i, t_i) for $i = 1,\ldots,n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs



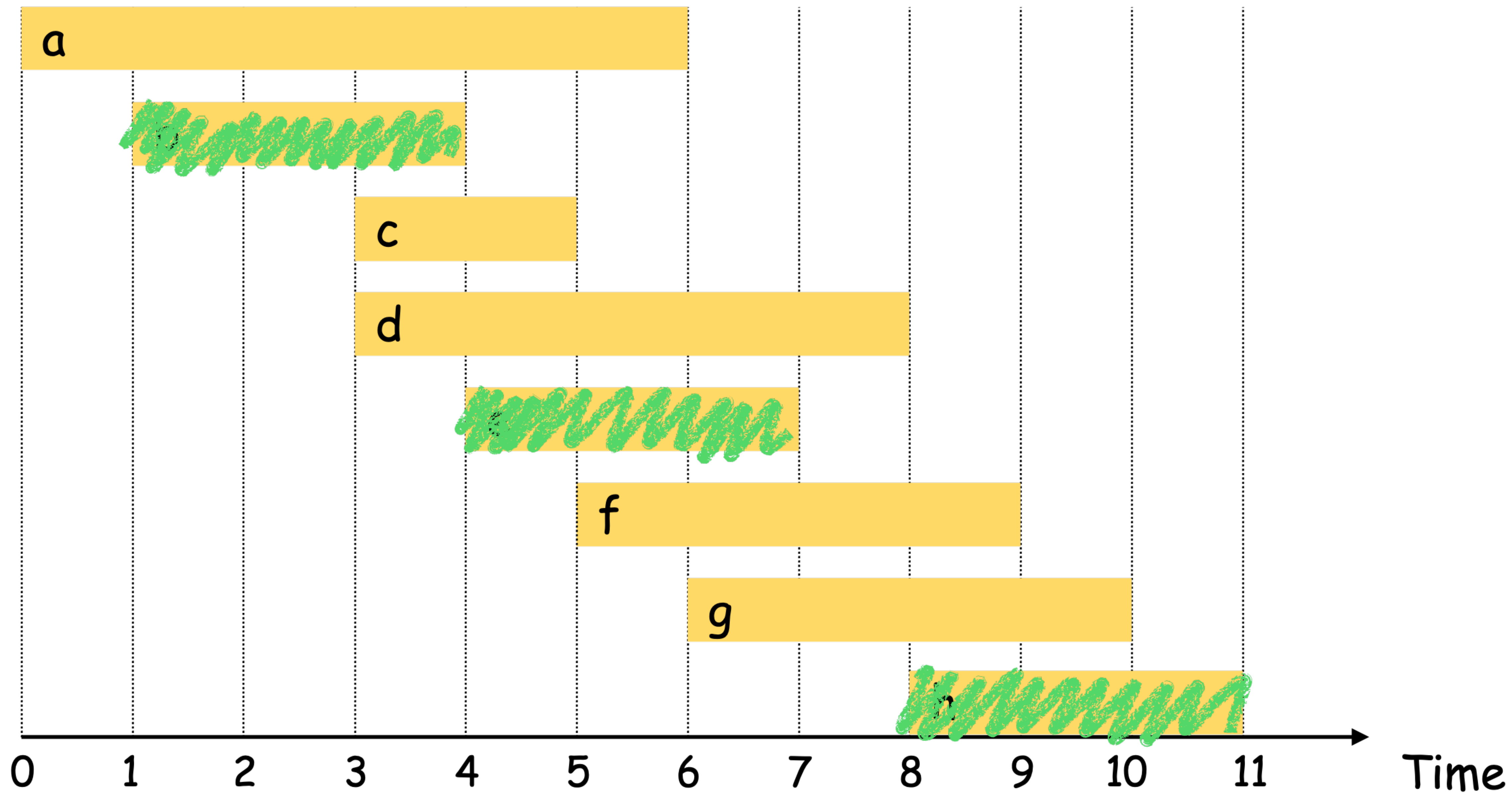
Interval scheduling

- **Input:** start and end times (s_i, t_i) for $i = 1,\ldots,n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs



Interval scheduling

- **Input:** start and end times (s_i, t_i) for $i = 1, \ldots, n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs

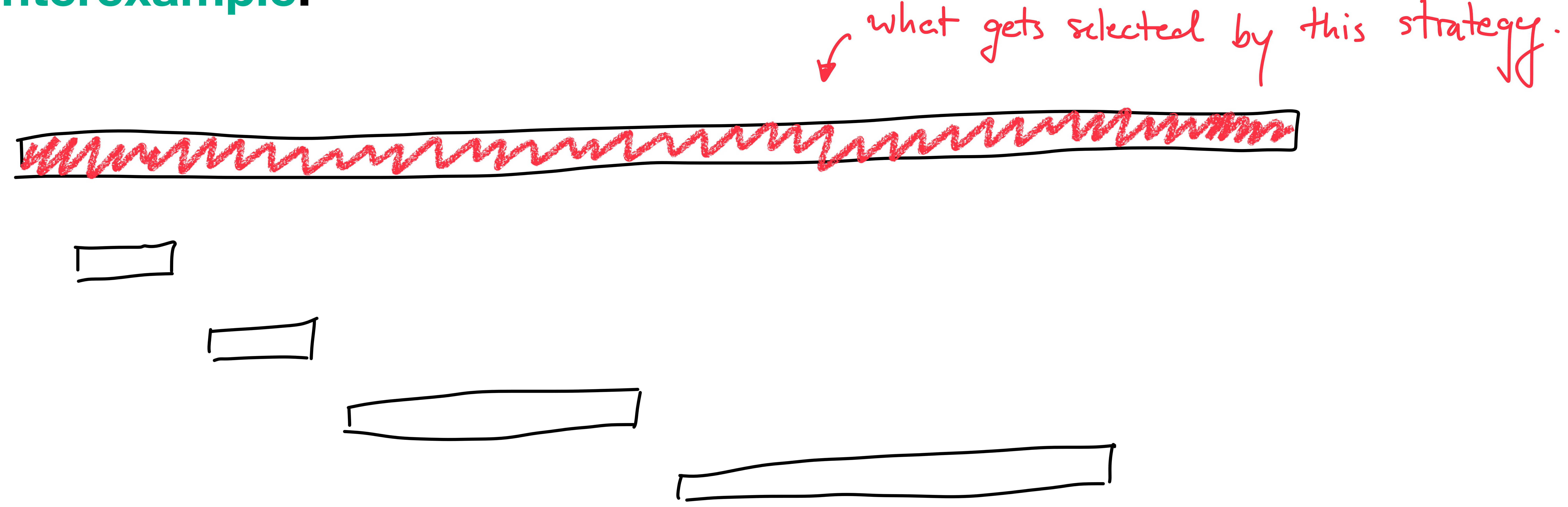


Interval scheduling

- **Input**: start and end times (s_i, t_i) for $i = 1,\ldots,n$ for n "jobs"
- **Output**: A maximal set of mutually compatible jobs
- **Algorithm**:
  - **Brute-force**: Iterate through all 2^n possible selections. Check in $O(n)$ time if selection is (a) feasible and (b) maximal.
  - **Greedy**: Decide a selection criteria and select jobs accordingly.

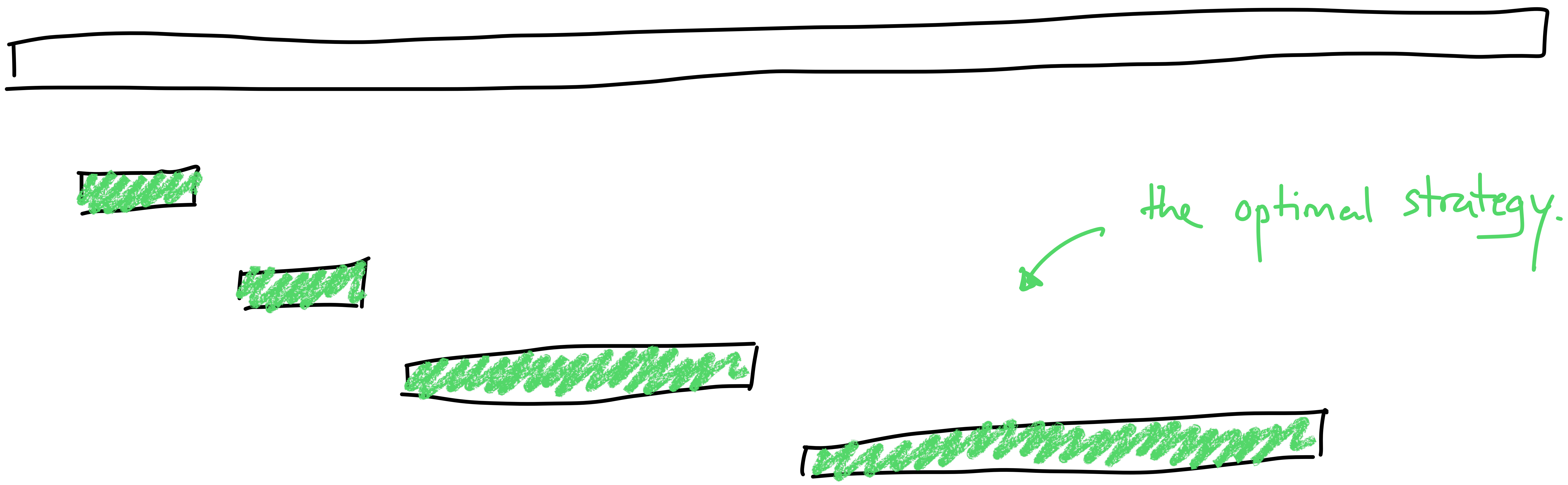
Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with earliest start time s_i of jobs not selected.
- **Counterexample:**



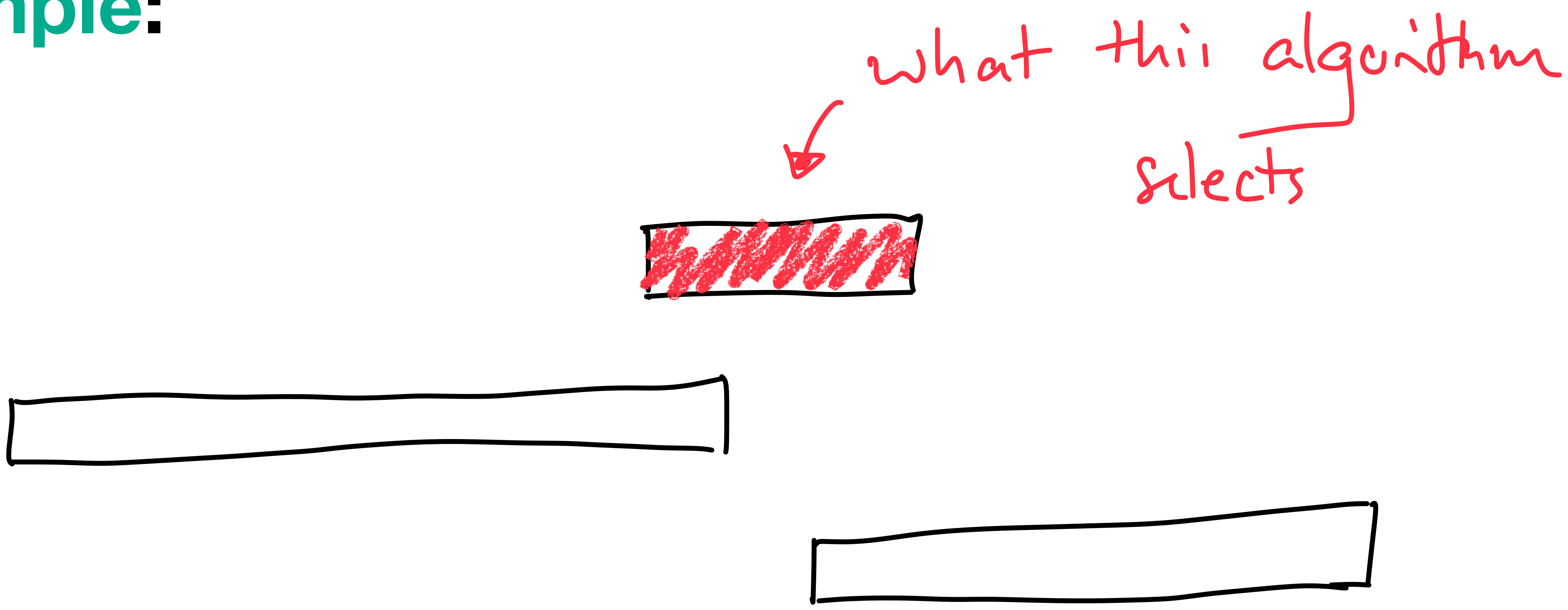
Greedy algorithms for interval scheduling

- **Algorithm**: Select the job with earliest start time s_i of jobs not selected.
- **Counterexample**:



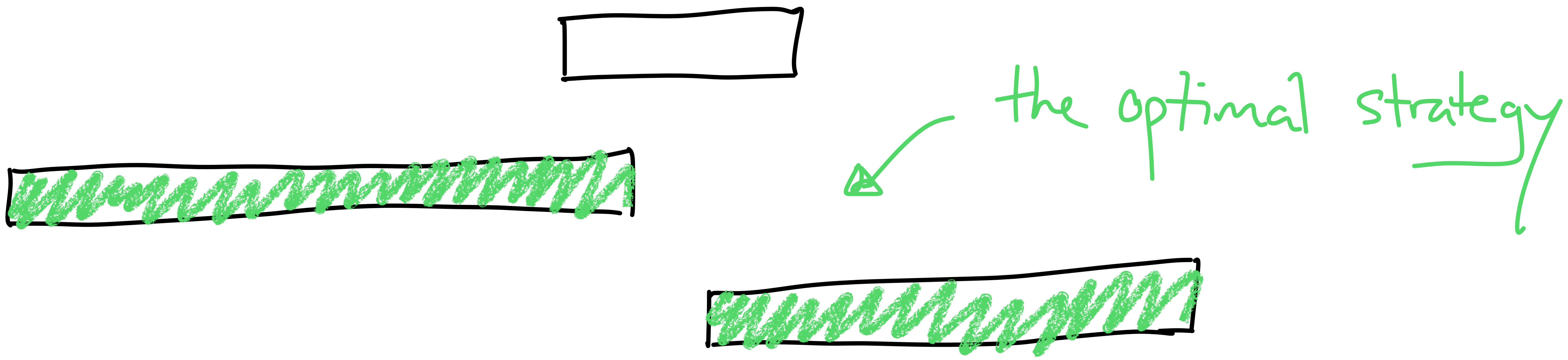
Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with shortest duration $t_i - s_i$ of jobs not selected.
- **Counterexample:**



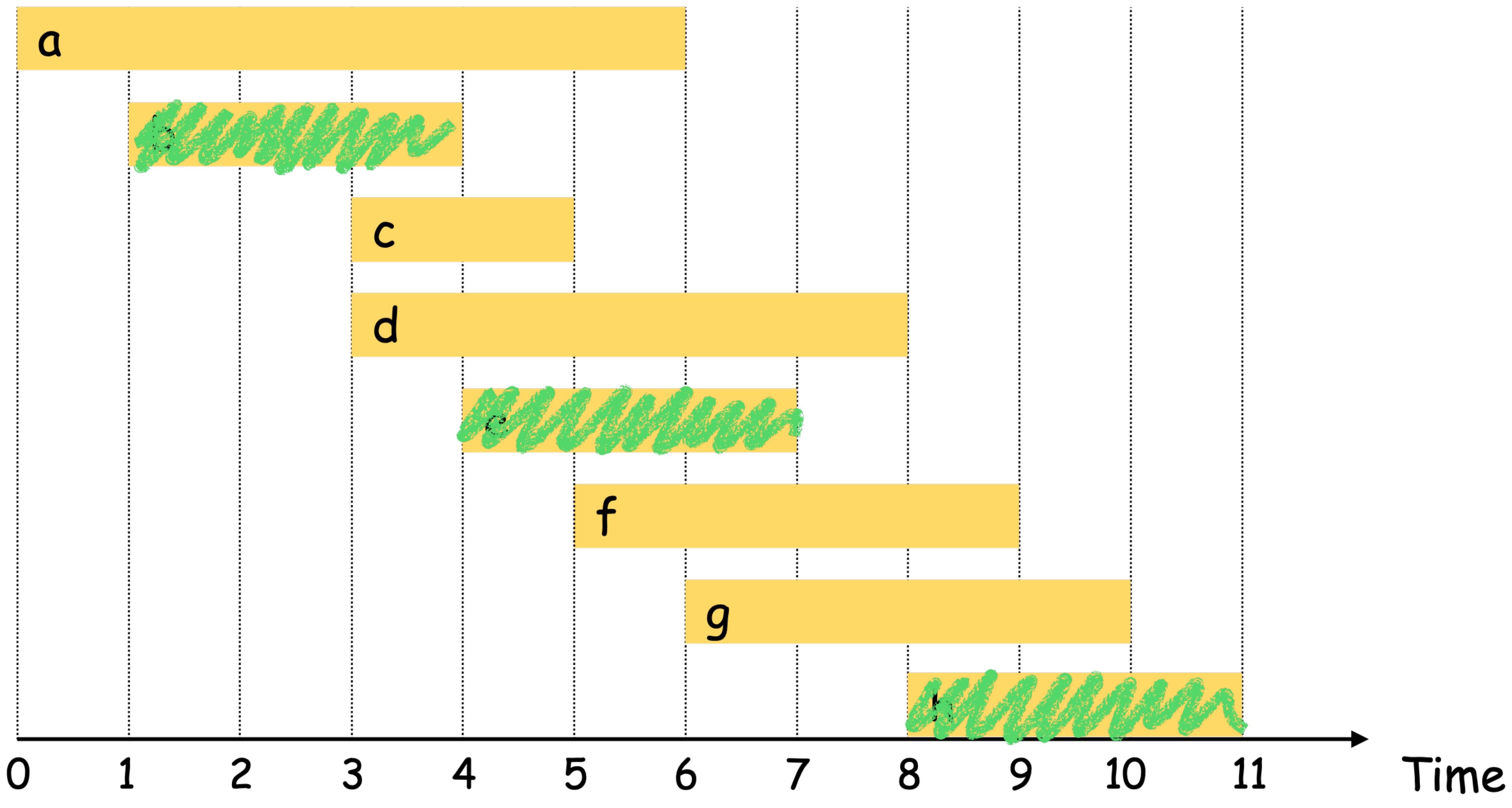
Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with shortest duration $t_i - s_i$ of jobs not selected.
- **Counterexample:**



Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with earliest ending t_i of jobs not selected.
- **Example:**



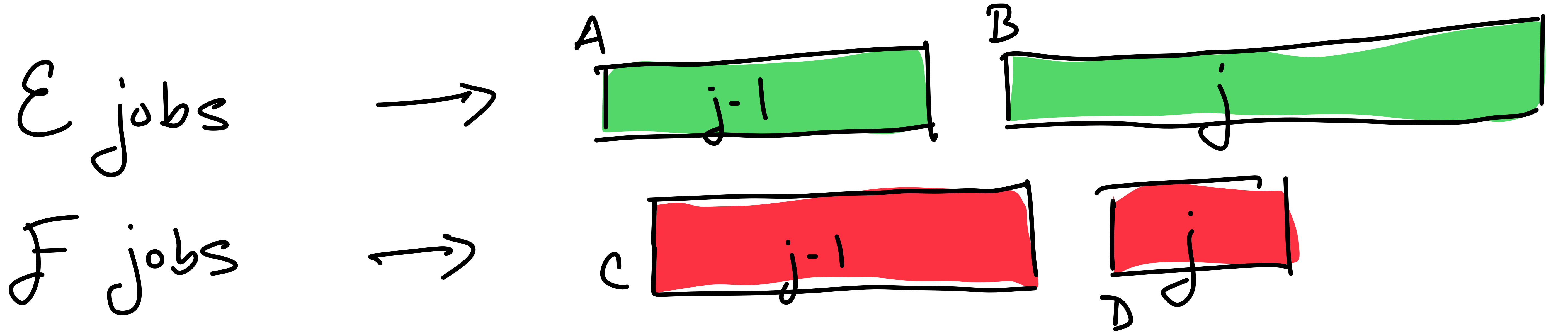
Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with earliest ending t_i of jobs not selected.
- **Proof of correctness:**
  - Let $\mathscr{E} \subseteq [n]$ be the set of jobs selected by algorithm and $\mathscr{F} \subseteq [n]$ be any other *feasible* set of jobs.
  - **Claim:** The j-th job in $\mathscr{E}$ ends at least before the j-th job in $\mathscr{F}$ ends.

Greedy algorithms for interval scheduling

- **Claim:** The j-th job in $\mathscr{E}$ ends at least before the j-th job in $\mathscr{F}$ ends.
- **Proof:**

Assume (for contradiction) that this is false and let j be the smallest counterexample. Picture:



Contradicts the def. of $\mathcal{E}$ as job D isn't selected but ends before job B.

Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with earliest ending t_i of jobs not selected.
- **Proof of correctness:**
  - Let $\mathscr{E} \subseteq [n]$ be the set of jobs selected by algorithm and $\mathscr{F} \subseteq [n]$ be any other *feasible* set of jobs.
  - **Claim:** The j-th job in $\mathscr{E}$ ends at least before the j-th job in $\mathscr{F}$ ends.
  - If $\mathscr{F}$ had more jobs than $\mathscr{E}$, we could have added the final job of $\mathscr{F}$ to $\mathscr{E}$, a contradiction to the def. of $\mathscr{E}$.
  - So, $\mathscr{E}$ has at least as many jobs as $\mathscr{F}$. True for all feasible $\mathscr{F}$, proving optimality.

Greedy algorithms for interval scheduling

- **Input**: start and end times (s_i, t_i) for $i = 1,\ldots,n$ for n "jobs"
- **Output**: A maximal set of mutually compatible jobs
- **Algorithm:** Select the job with earliest ending t_i of jobs not selected.
  - **Details:** Sort the jobs by earliest end time t_i. Keep track of current end time of selected jobs T. Add new job (s_i, t_i) if $s_i \geq T$ and update $T \leftarrow t_i$.
  - **Runtime:** Sorting + linear time to create list of jobs. $O(n \log n) + O(n) = O(n \log n)$.

Greedy algorithms

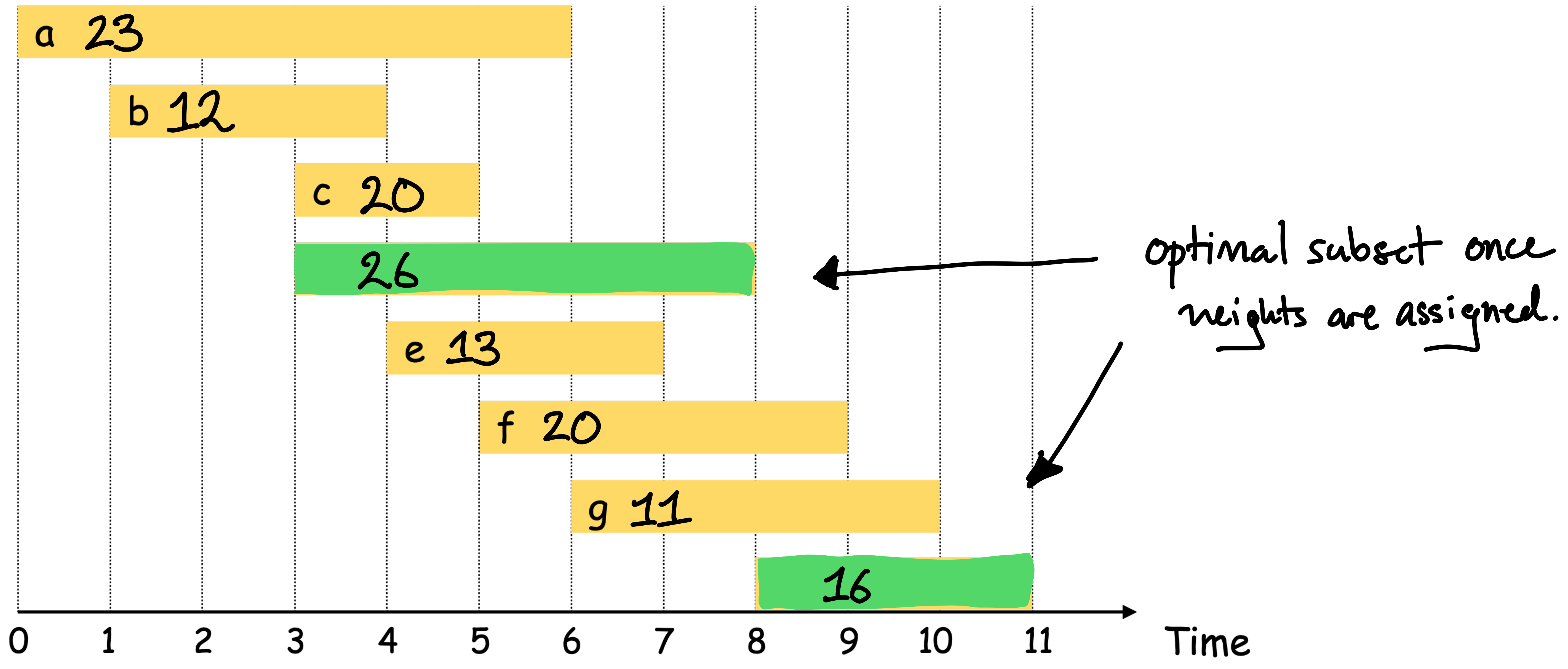
- Myopic style of argument that makes decisions based on current information. Does not “look ahead”.
- Greedy algorithms tend to only work when there is some kind of special structure to the problem.
- When they do work, they are very efficient.
- When they don’t work, they often give very good approximation algorithms!

Weighted interval scheduling

- Same problem as interval scheduling except each job has a value w_i. Want to optimize sum of weights of jobs selected.
  - Example: Scheduling rooms for a conference except some speeches pay more.
  - Example: $w_i = t_i - s_i$. The value is the length of the job.

Weighted interval scheduling

- **Input**: start and end times (s_i, t_i) and weights w_i for $i = 1, \ldots, n$ for n "jobs"
- **Output**: A set of mutually compatible jobs of maximal weight sum



Weighted interval scheduling

- **Input**: start and end times (s_i, t_i) and weights w_i for $i = 1,\ldots,n$ for n "jobs"
- **Output**: A set of mutually compatible jobs of maximal weight sum
- If all weights $w_i = 1$, then this is regular interval scheduling
- In general, we need a different technique: **Dynamic Programming**
    - Build up solution from a table of precomputed solutions to smaller problems

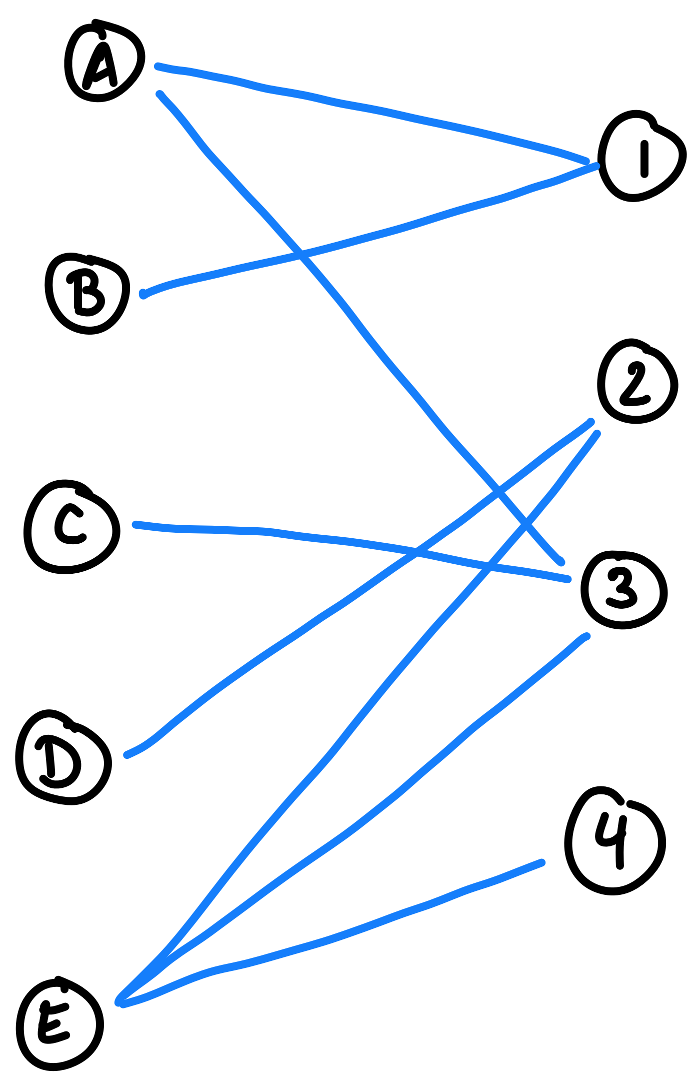
Dynamic programming

Principal properties

- **Optimal substructure**:
  - The optimal value of the problem can easily be obtained given the optimal values of subproblems.
  - In other words, there is a recursive algorithm for the problem which would be fast if we could just skip the recursive steps.
- **Overlapping subproblems**:
  - The subproblems share sub-subproblems.
  - In other words, if you actually ran that naïve recursive algorithm, it would waste a lot of time solving the same problems over and over again.

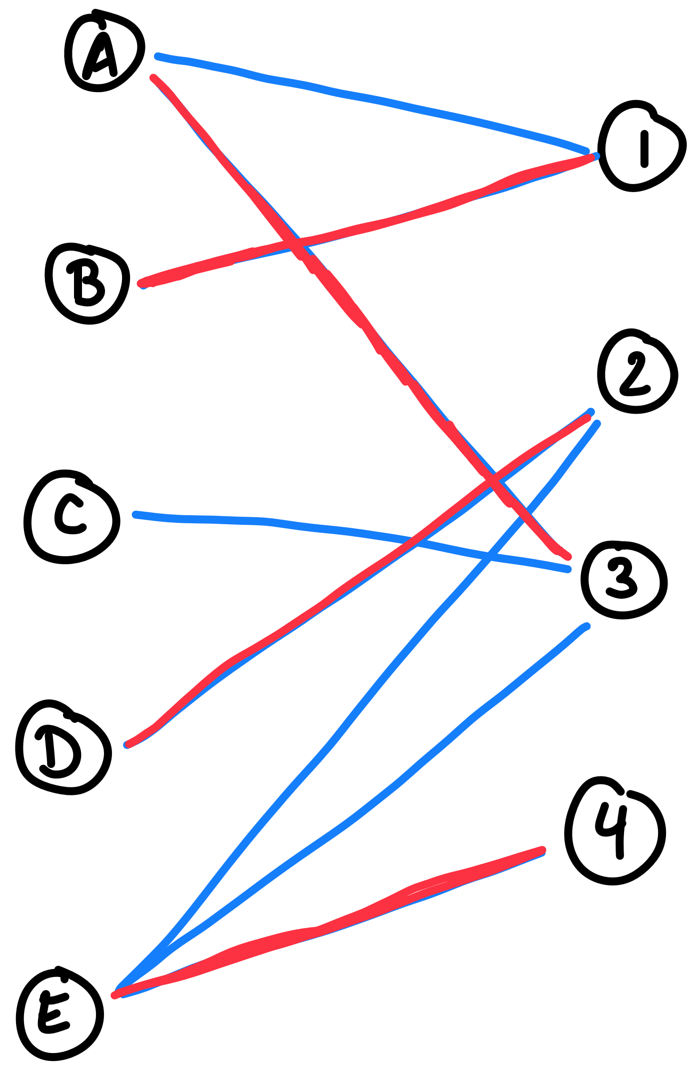
Bipartite matching

- A graph $G = (V, E)$ is bipartite iff
  - V, the vertices, have two disjoint parts $V = X \sqcup Y$.
  - Every edge $e \in E$ connects a vertex $x \in X$ and a $y \in Y$.
- A set $M \subseteq E$ is a **matching** if no two edges in M share a vertex.
- **Goal:** Find a matching of maximal size given input bipartite G.



Bipartite matching

- A graph $G = (V, E)$ is bipartite iff
  - V, the vertices, have two disjoint parts $V = X \sqcup Y$.
  - Every edge $e \in E$ connects a vertex $x \in X$ and a $y \in Y$.
- A set $M \subseteq E$ is a **matching** if no two edges in M share a vertex.
- **Goal:** Find a matching of maximal size given input bipartite G.

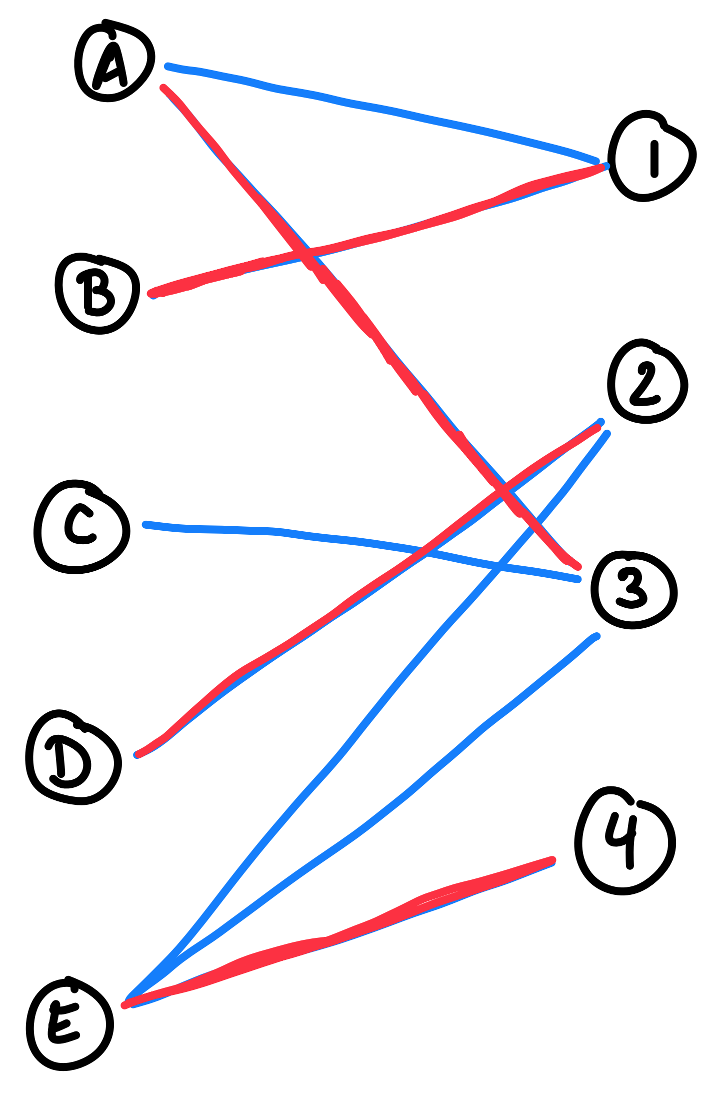


Bipartite matching

- A set $M \subseteq E$ is a **matching** if no two edges in M share a vertex.

- **Goal:** Find a matching of maximal size given input bipartite G.

- Differences from stable matching:
    - Limited set of possible partners for each vertex.
    - Sides may not be the same size.
    - No notion of stability, matching everything may be impossible.

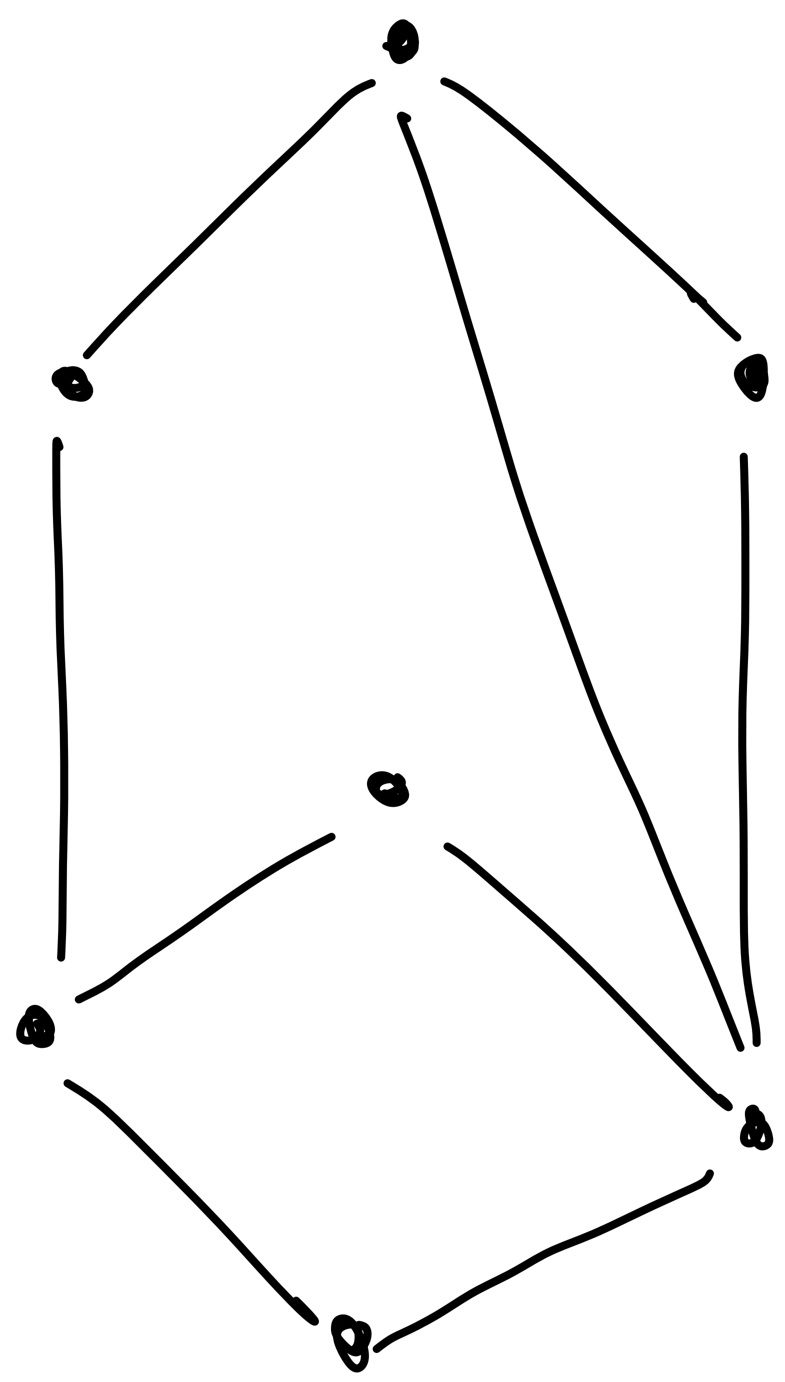


Bipartite matching

- **Applications**:
  - X represent visitors to websites, Y represent servers they access
  - X represents professors, Y represents courses
  - X represents taxi riders, Y represents taxis
- If $|X| = |Y| = n$, then G has a "perfect matching" if the matching is size n
- **Finding bipartite matchings**:
  - Polynomial-time algorithm using "augumentation" technique
  - Solved via general class of *network flow* problems and algorithms

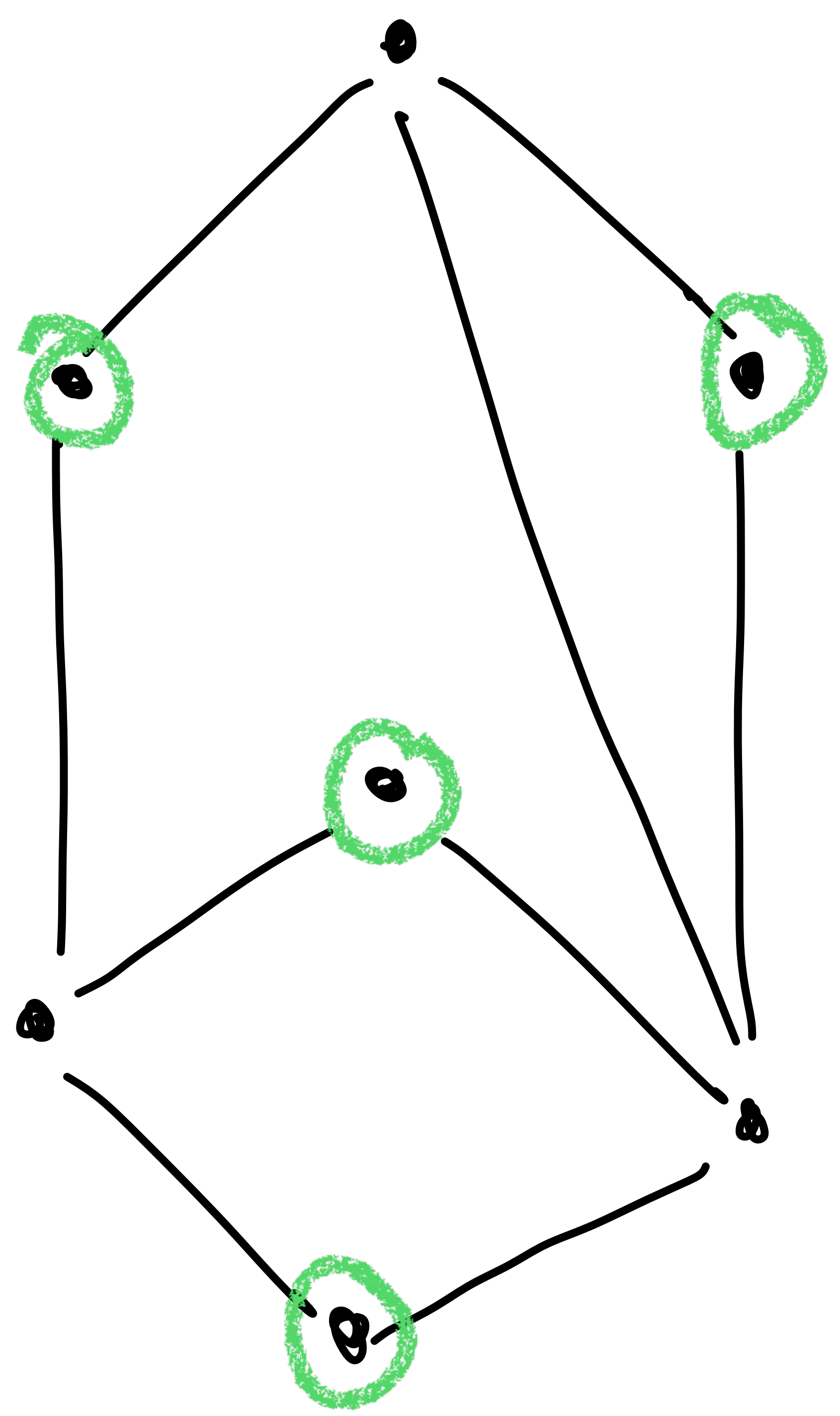
Independent set

- Given a graph $G = (V, E)$, a subset $I \subseteq V$ is *independent* if there are no edges between vertices of I
- **Input**: graph $G = (V, E)$
- **Output**: Find an independent set of maximal size.
- Models selecting set of non-conflicting scenarios.
  - Example: Assigning frequencies in radio networks
  - Example: Scheduling exams with edges represent classes that share a student.



Independent set

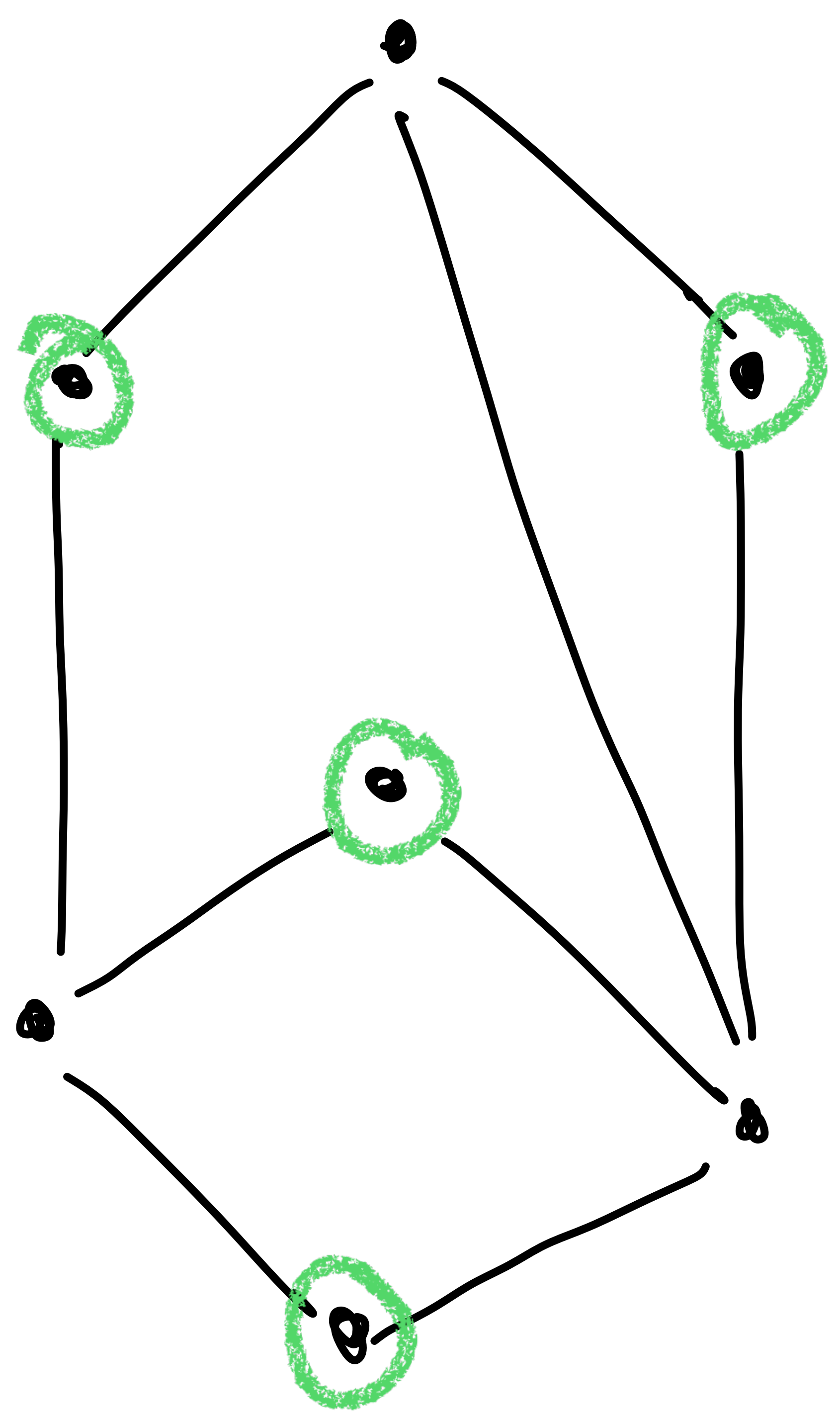
- Given a graph $G = (V, E)$, a subset $I \subseteq V$ is *independent* if there are no edges between vertices of I
- **Input**: graph $G = (V, E)$
- **Output**: Find an independent set of maximal size.
- Models selecting set of non-conflicting scenarios.
  - Example: Assigning frequencies in radio networks
  - Example: Scheduling exams with edges represent classes that share a student.



Independent set

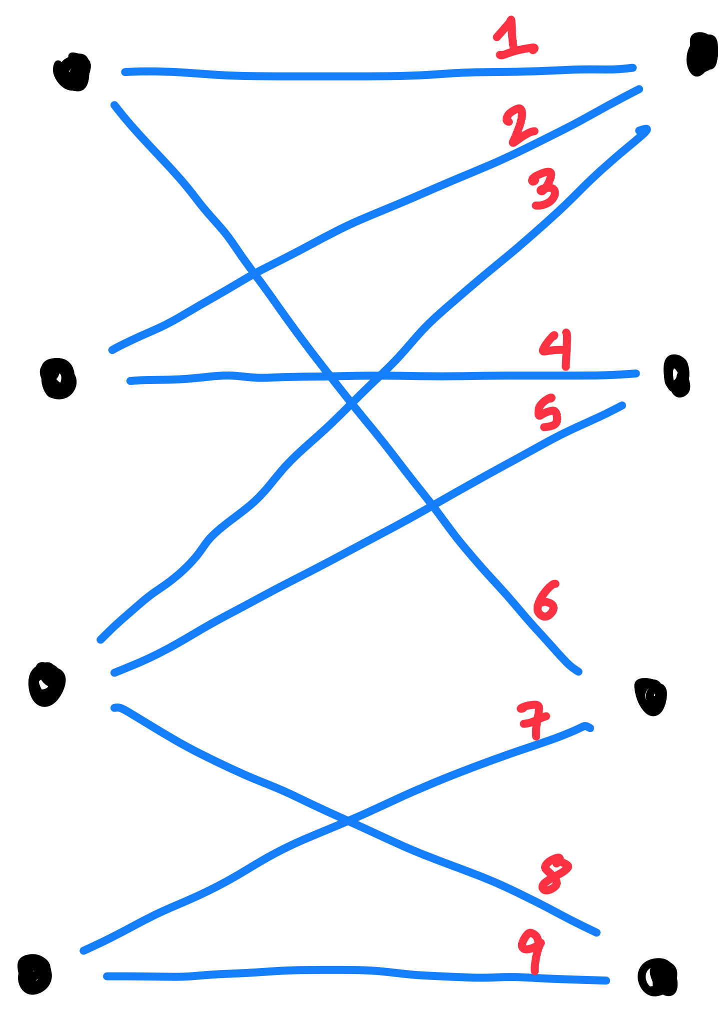
Generalizes many problems

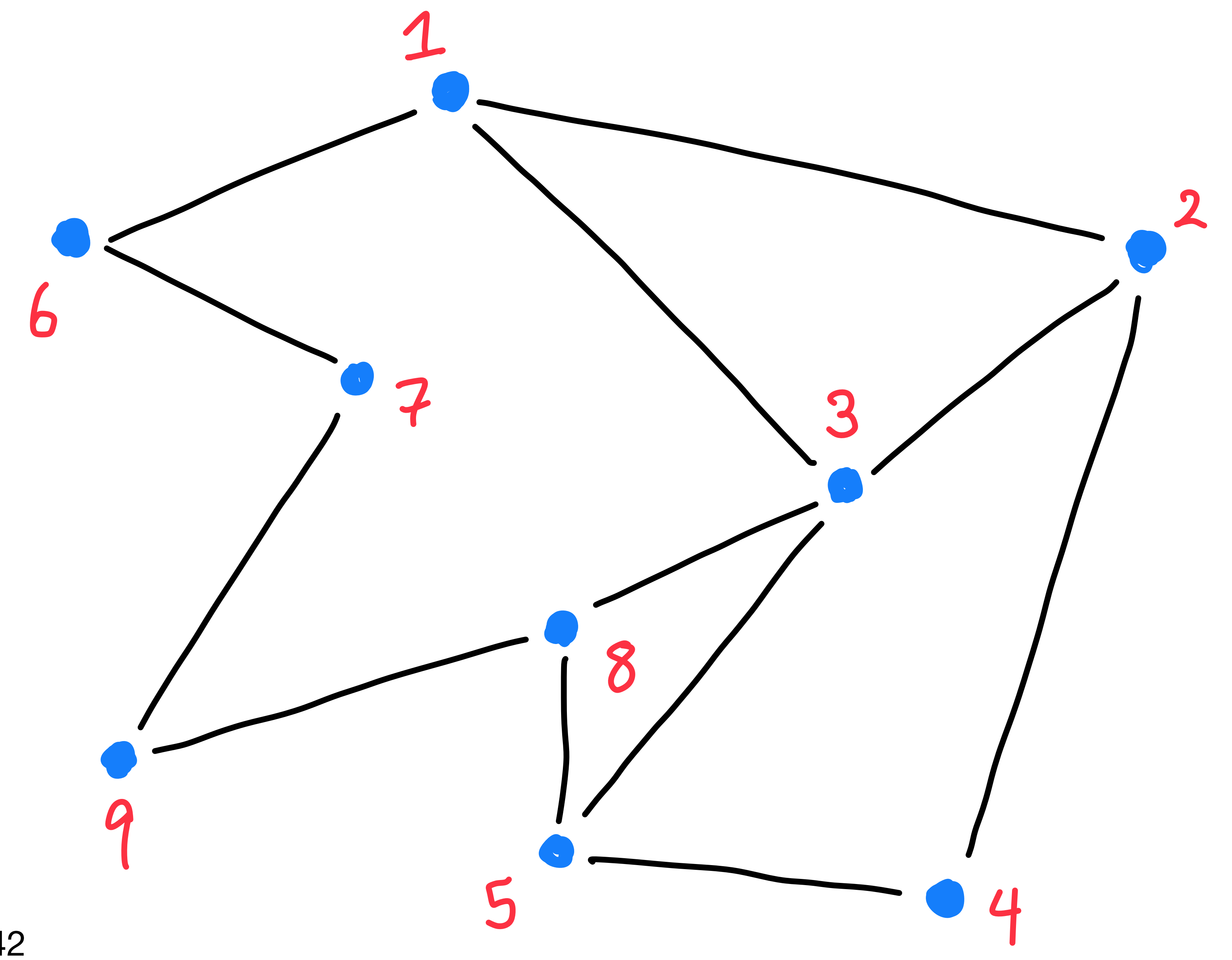
- Generalizes **Interval Scheduling** and **Bipartite Matching**
- **Interval Scheduling**
  - Vertices correspond to the jobs
  - Edges between jobs that *overlap*
- **Bipartite matching**
  - Given graph G create a new graph L_G called the line-graph
  - Independent set in L_G corresponds to a bipartite matching in G



Line Graph Construction

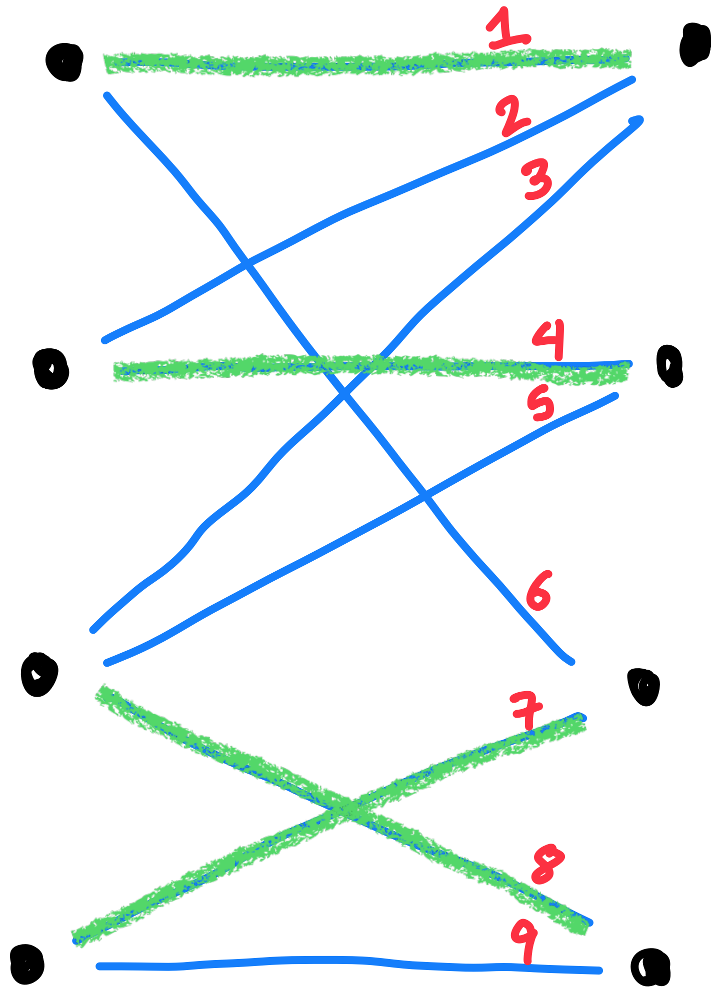
- From a graph $G = (V, E)$ to a new graph $G' = (V', E')$ where $V' = E$
- An edge exists in E' if two vertex/edges $e_1, e_2 \in E = V'$ share a vertex in V.
- **Picture:**

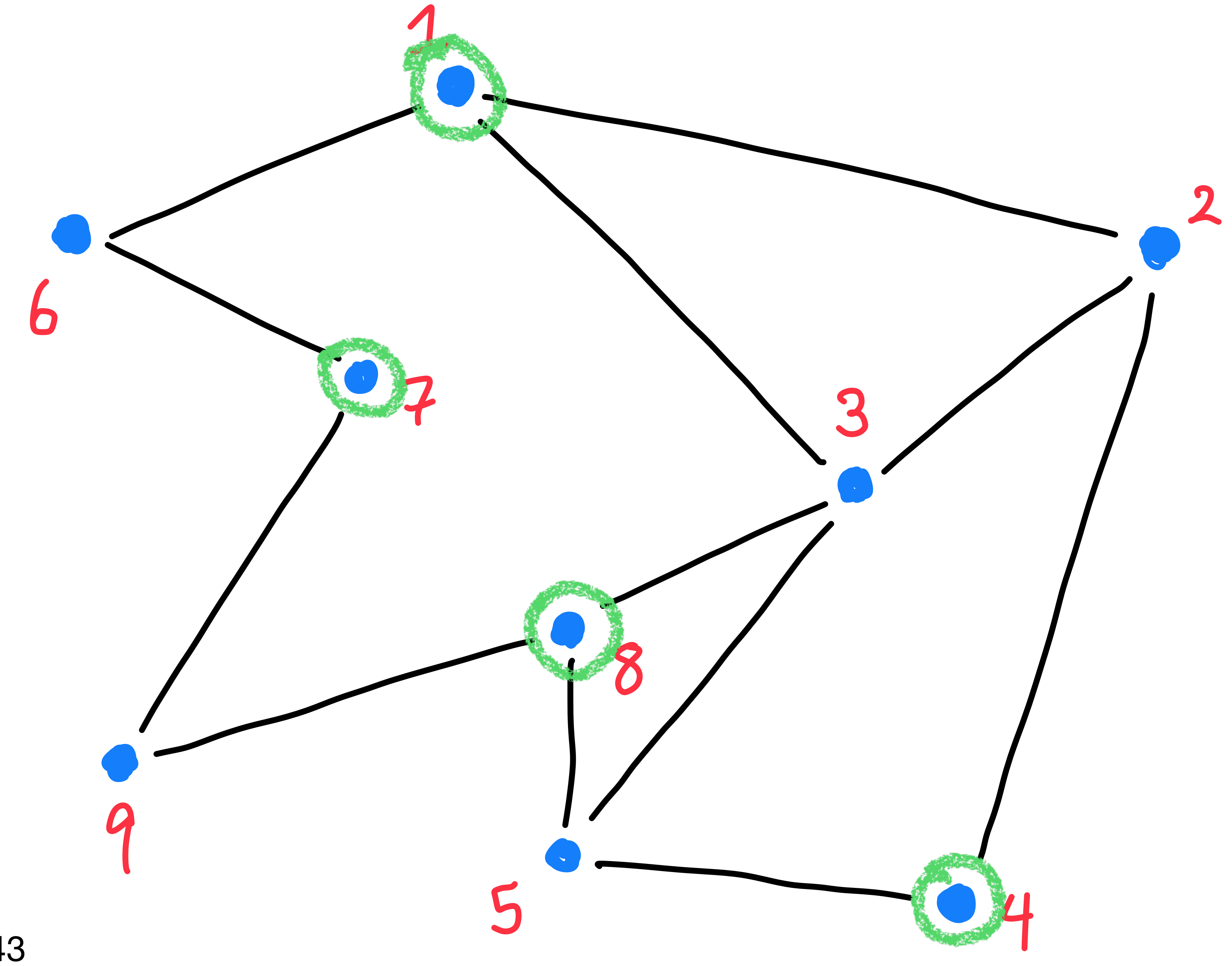




Line Graph Construction

- From a graph $G = (V, E)$ to a new graph $G' = (V', E')$ where $V' = E$
- An edge exists in E' if two vertex/edges $e_1, e_2 \in E = V'$ share a vertex in V.
- **Picture:**



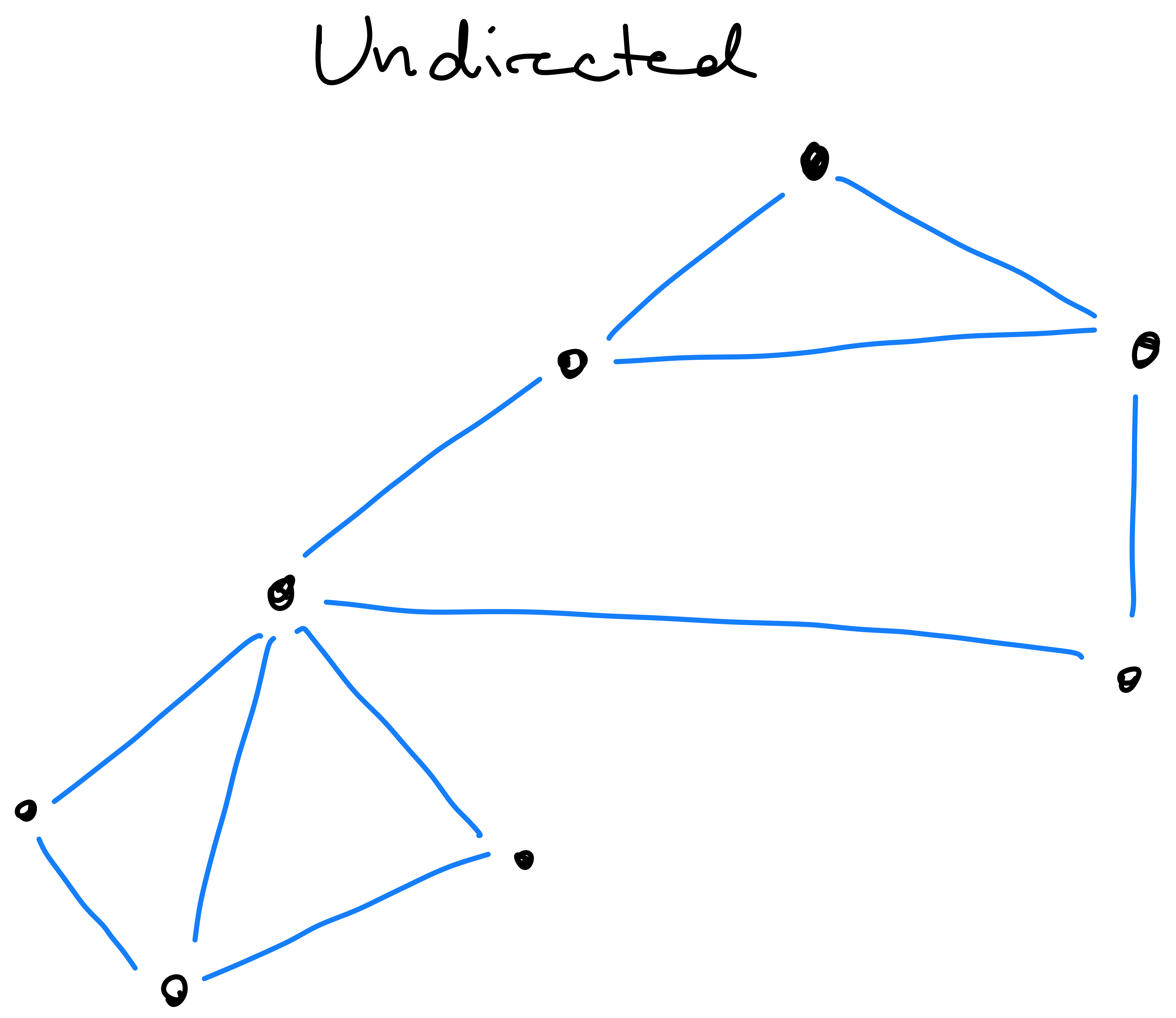


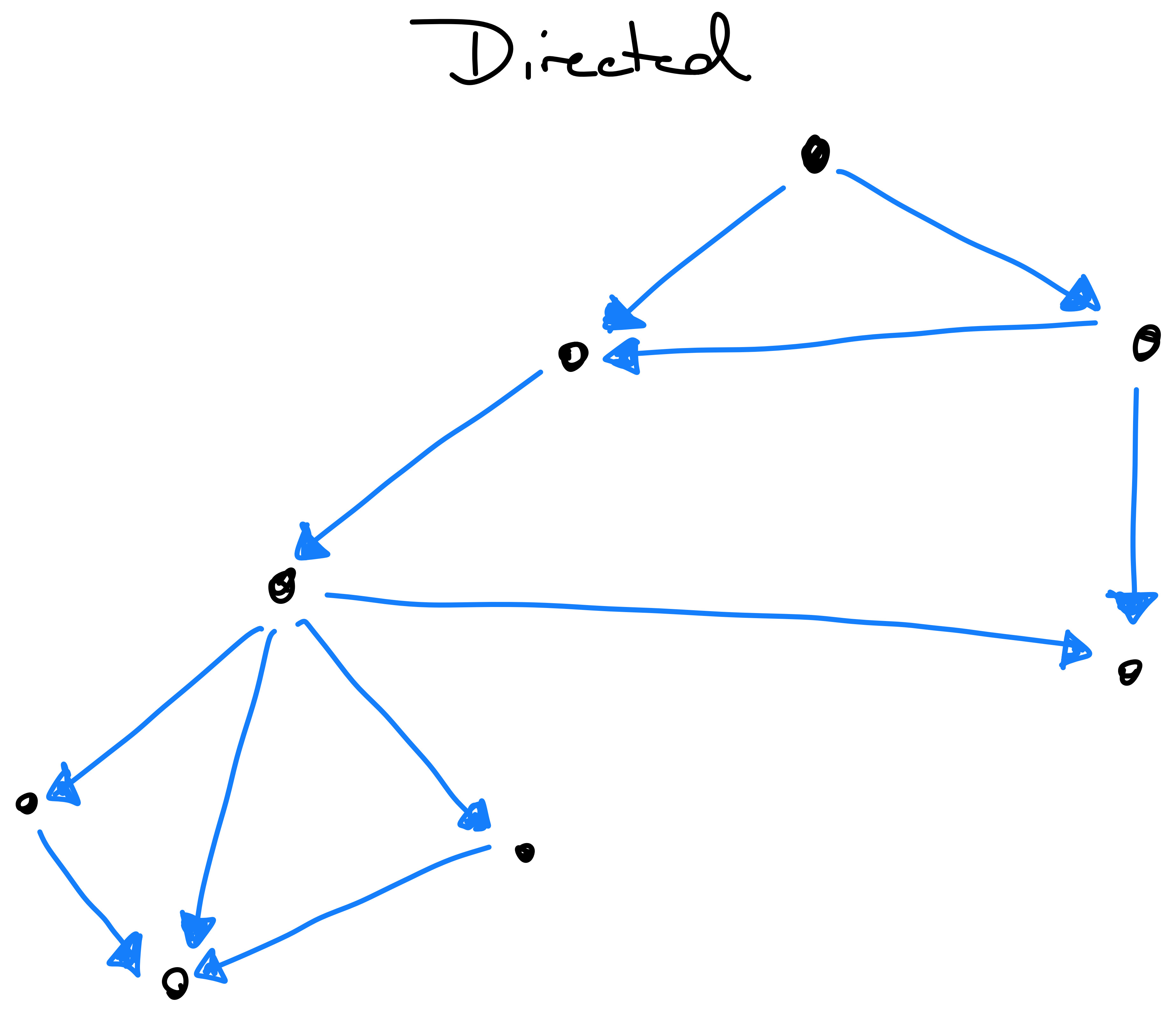
Hardness of independent set

- There is no known polynomial-time algorithm for independent set (even decision)
  - **Input:** Graph G and integer $k \geq 0$
  - **Output**: If there exists $I \subseteq V$ independent set such that $|I| \geq k$
- Easy to check in poly time if a solution I is both (a) valid and (b) at least size k
- Therefore the problem is in NP - **non-deterministic polynomial time.**
- Independent set is NP-complete, the hardest problem in NP.

Graph traversal

Graph search and traversal





Graph search and traversal

- Used to discover the structure of a graph
- "Walk" from a fixed starting vertex s ("the source") to find all the vertices reachable from s
- **Generic traversal algorithm.**
  - **Input:** Graph G and vertex $s \in V$
  - **Find:** set $R \subseteq V$ reachable from s

Reachable(s):

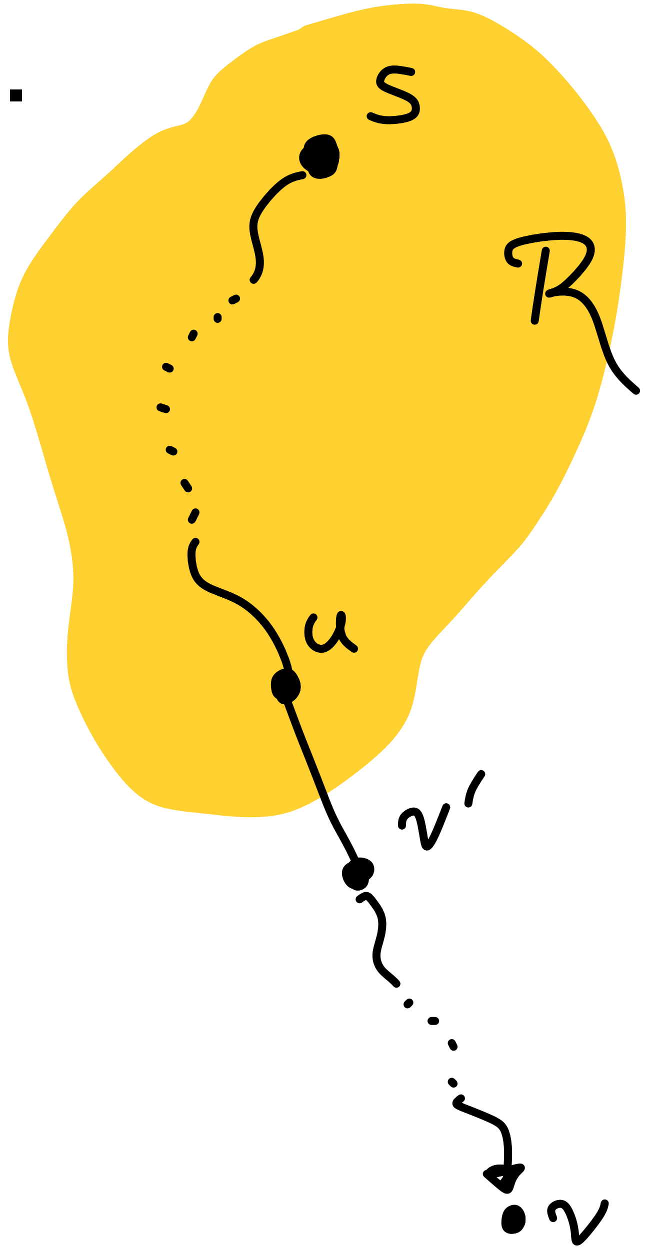
$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
  Add v to R: $R \leftarrow R \cup \{v\}$.
return R

Generic graph traversa

Reachable(s):

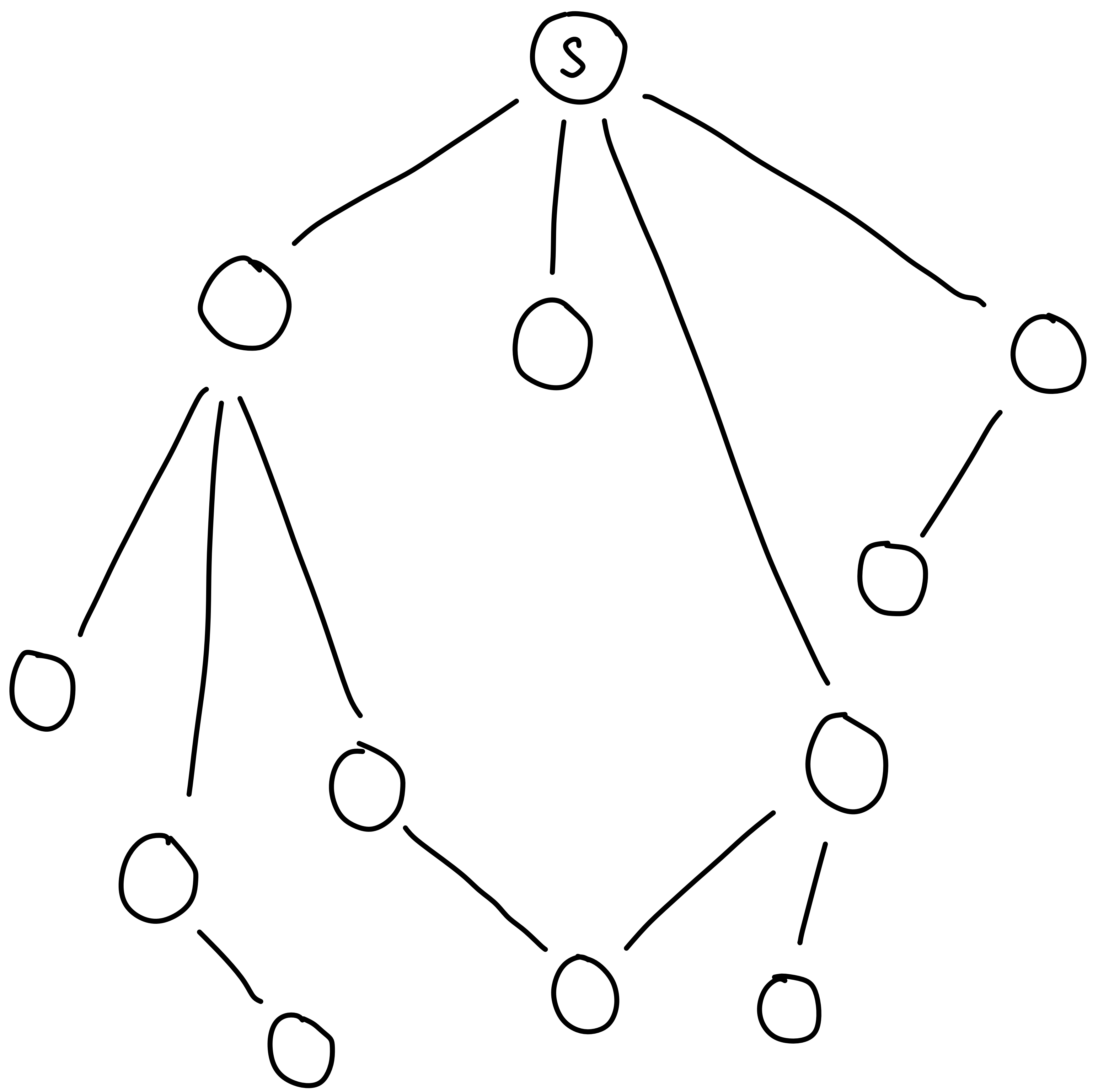
$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
    Add v to R: $R \leftarrow R \cup \{v\}$.
return R

- **Claim:** R is exactly the set of reachable vertices.
- **Proof:** We show both directions. (1): every vertex in R is reachable. (2): every reachable is in R.
  - **Direction 1**. For $v \in R$, there is a path $s \rightsquigarrow v$. Proved by induction on the generic graph traversal algorithm: If we added v by edge $(u, v) \in R \times (V \backslash R)$ then $s \rightsquigarrow u \rightarrow v$.
  - **Direction 2**. Assume (for $\perp$), there is a vertex v that is reachable but not $v \notin R$.
    - Let $p =$ the path $s \rightsquigarrow v$ and let v' be the **first** vertex on p such that $v' \notin R$.
    - Then u, the predecessor of v', satisfies $u \in R$ and $(u, v') \in R \times (V \backslash R)$.
    - Contradicts the definition of the generic graph traversal.



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

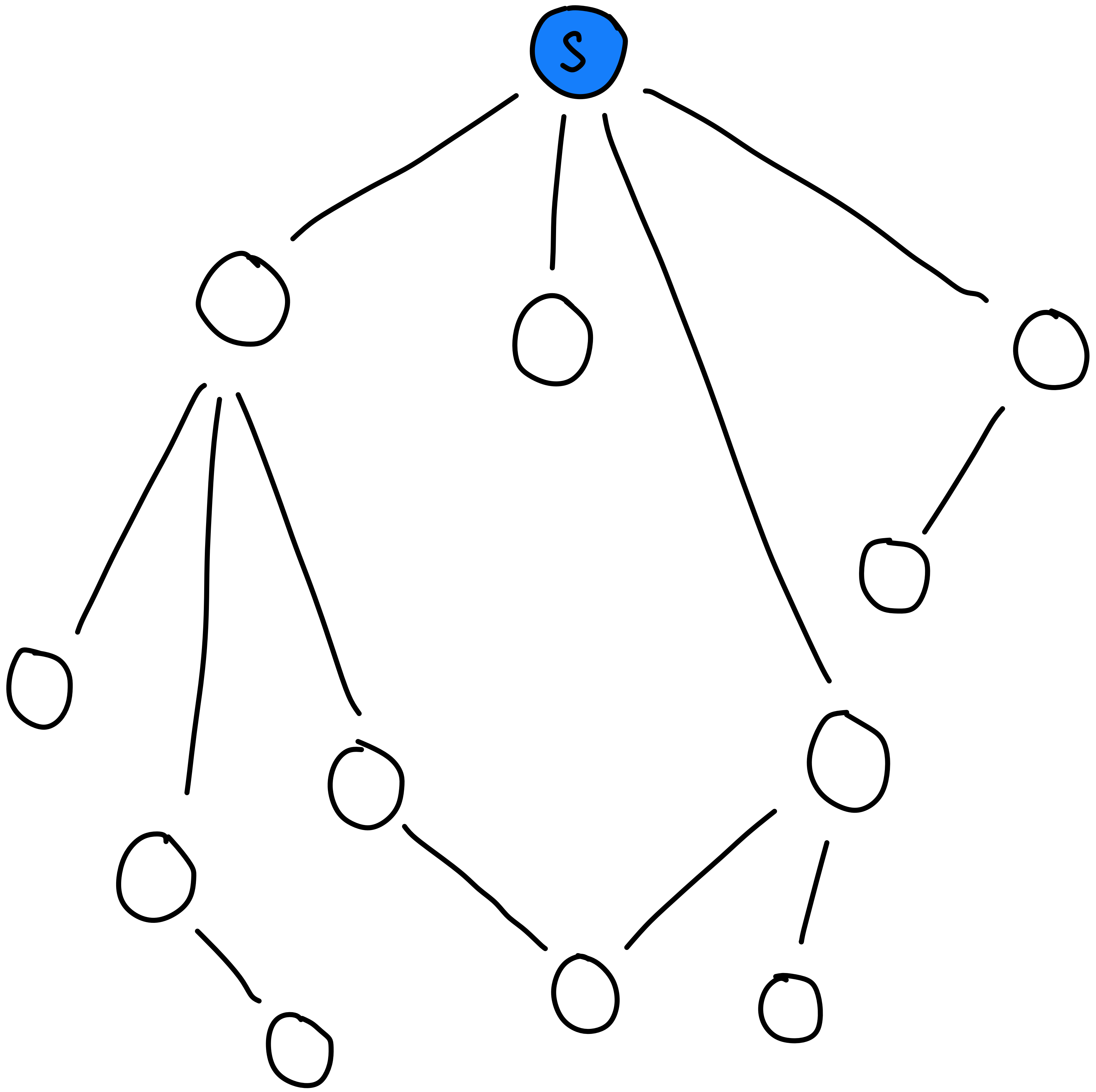


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

queue Q

s

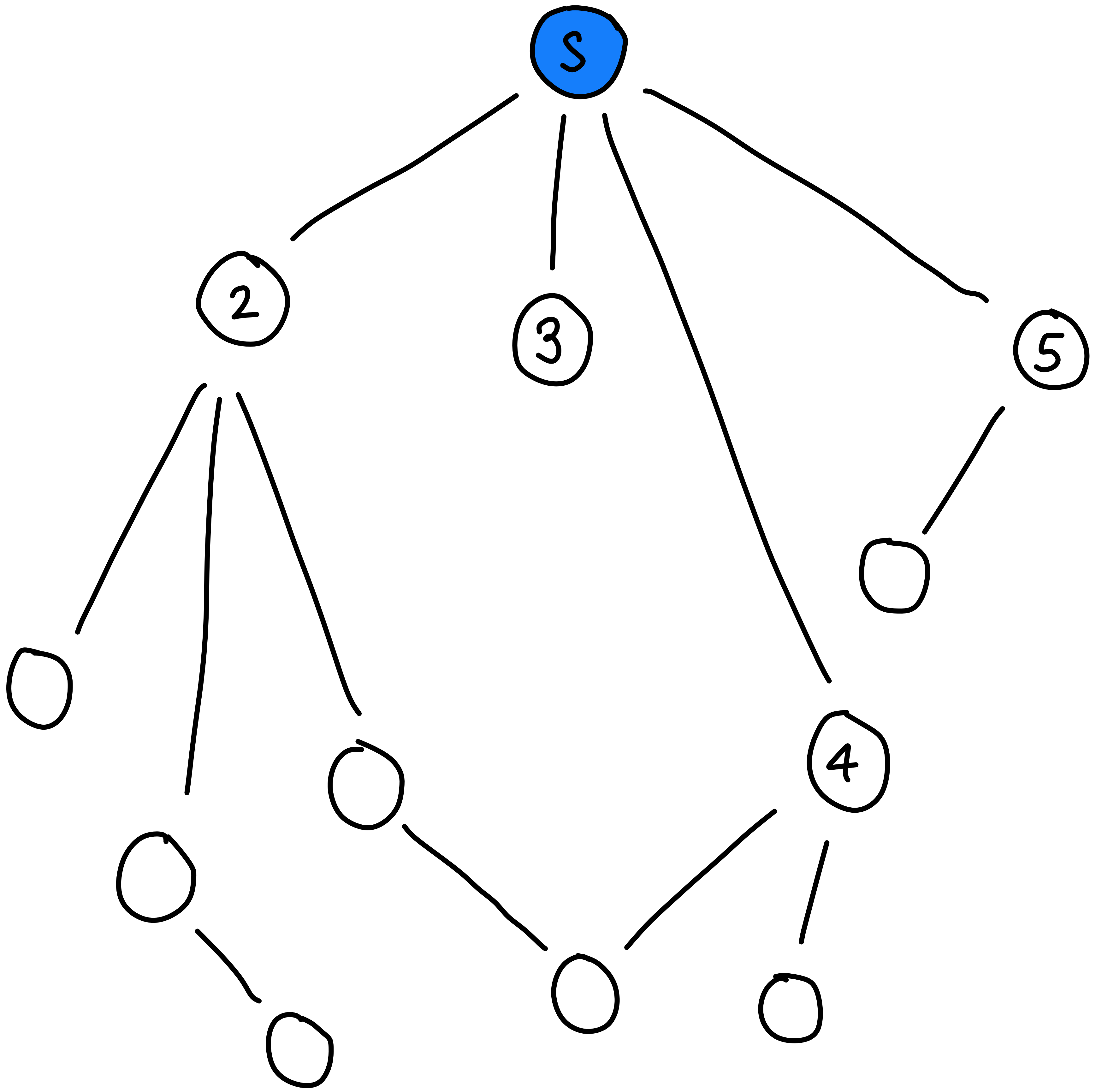


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
2
3
4
5

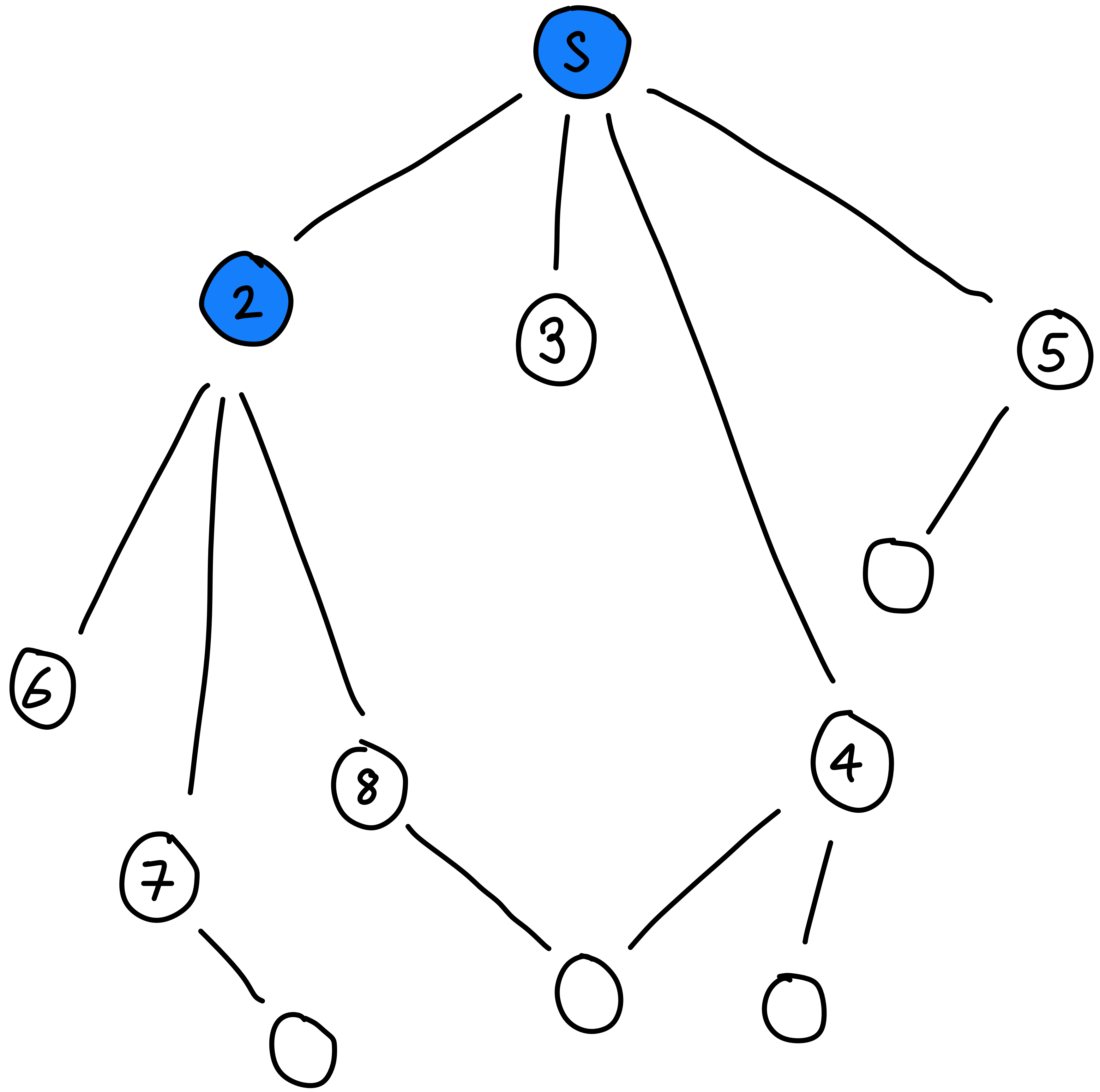


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
3
4
5
6
7
8

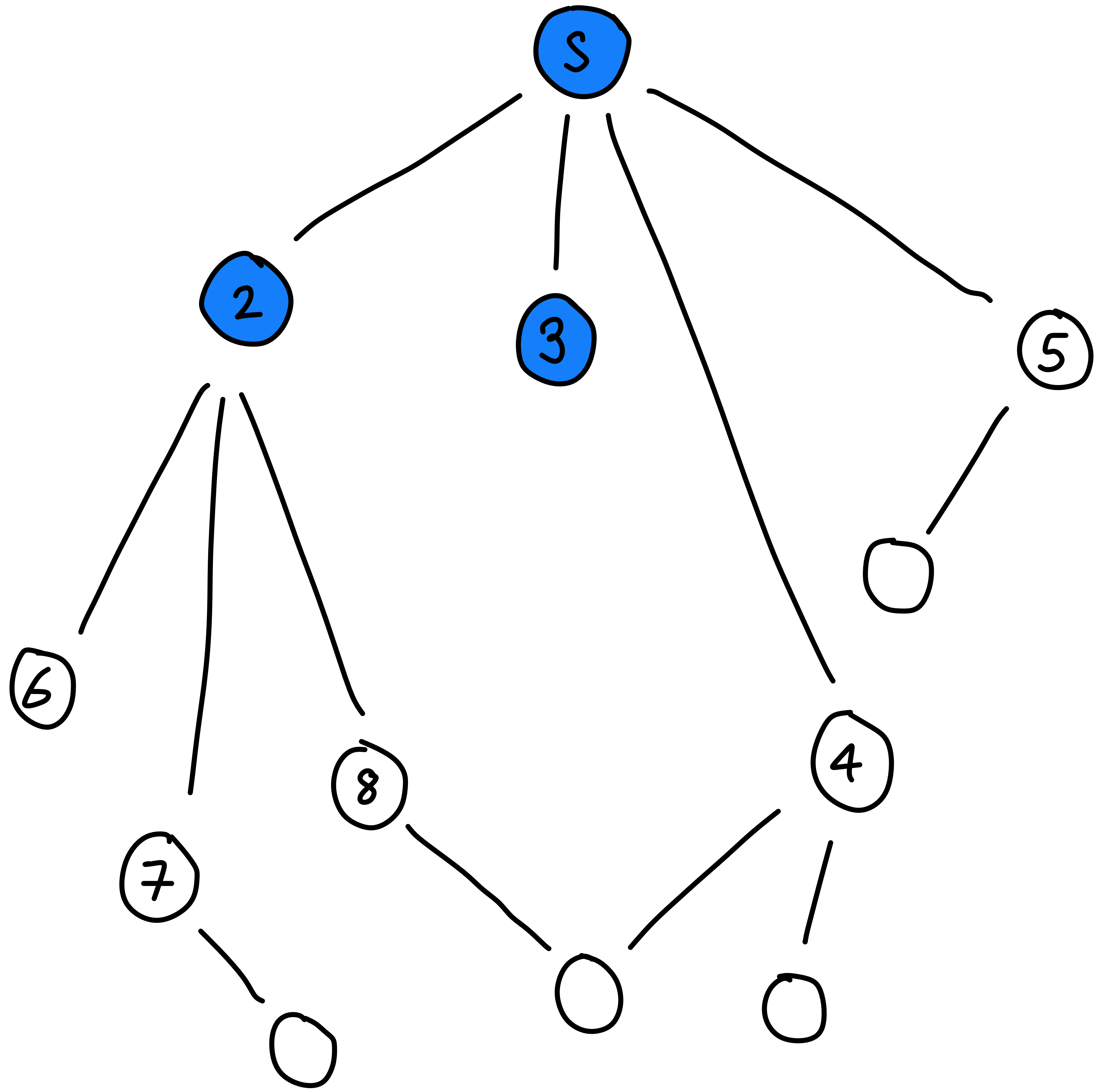


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
4
5
6
7
8

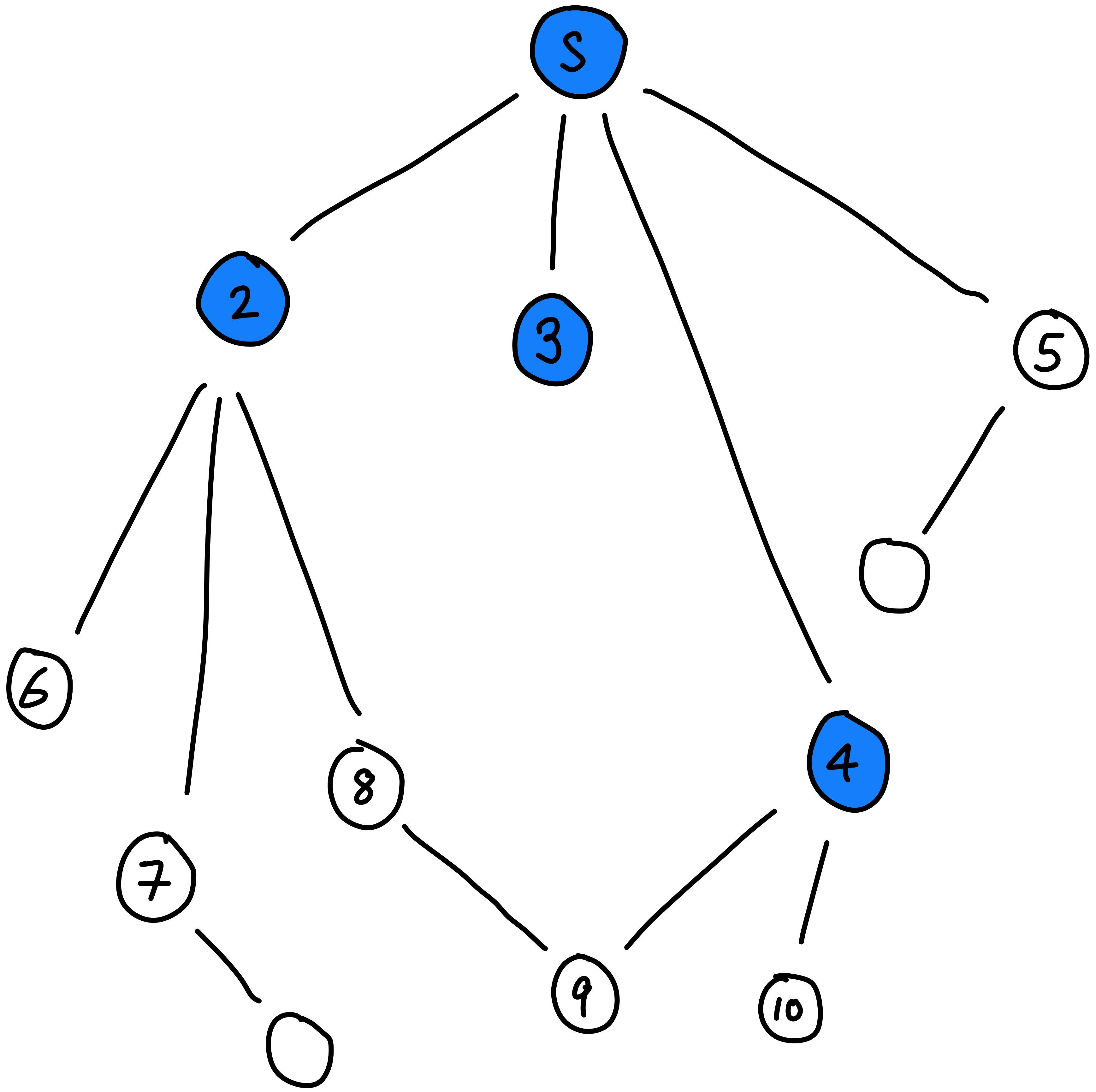


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
5
6
7
8
9
10

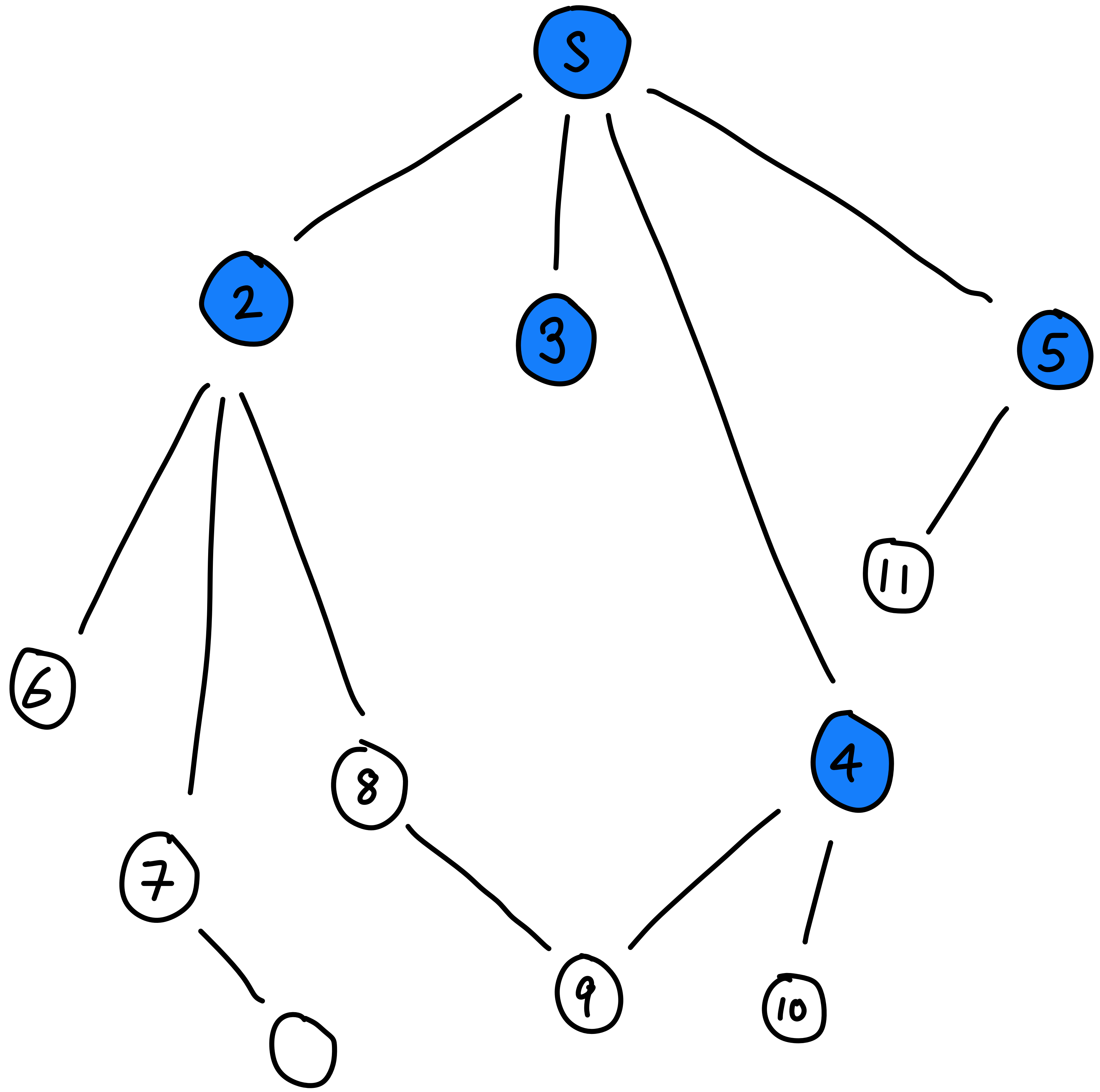


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
6
7
8
9
10
11

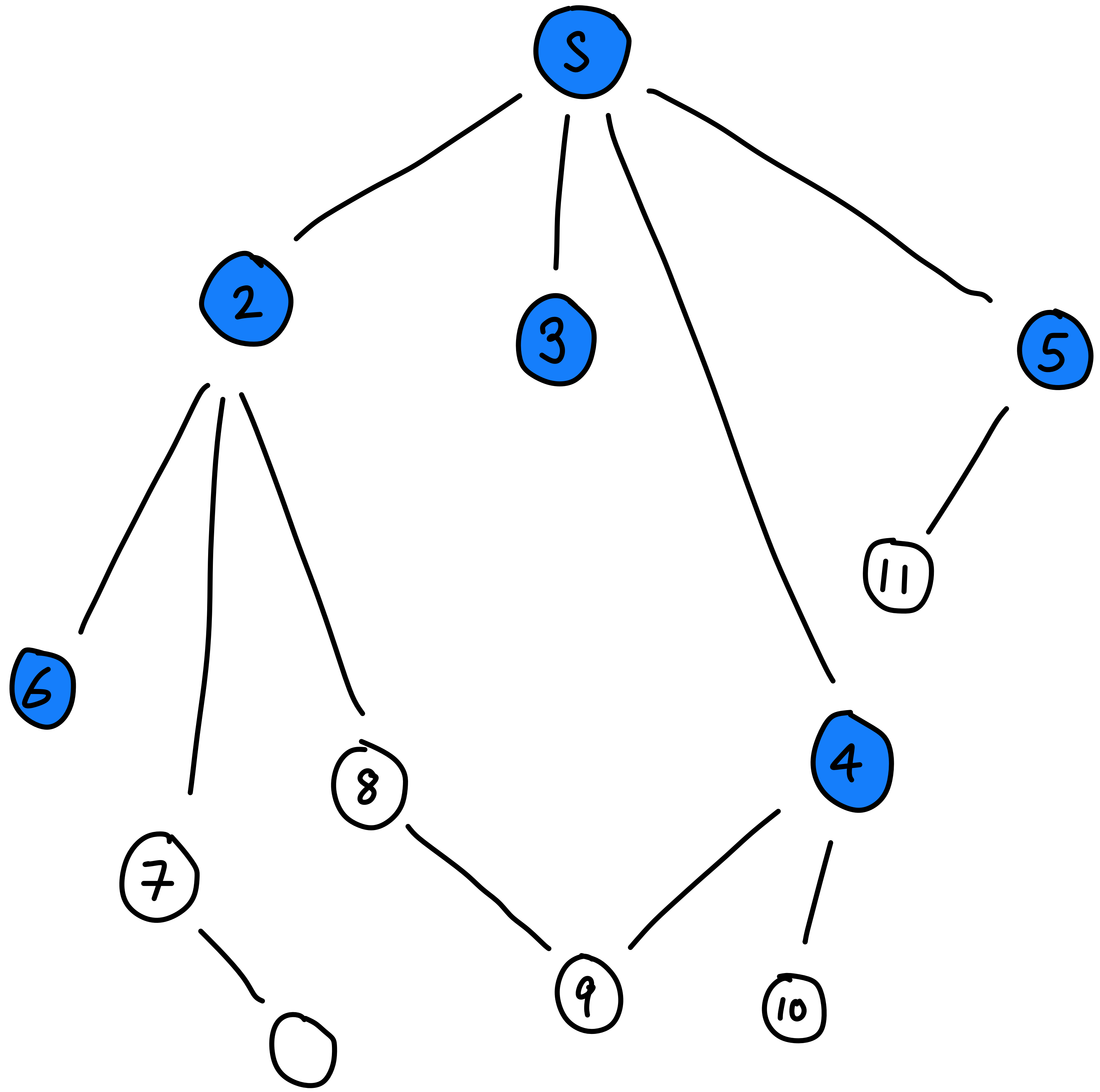


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
7
8
9
10
11

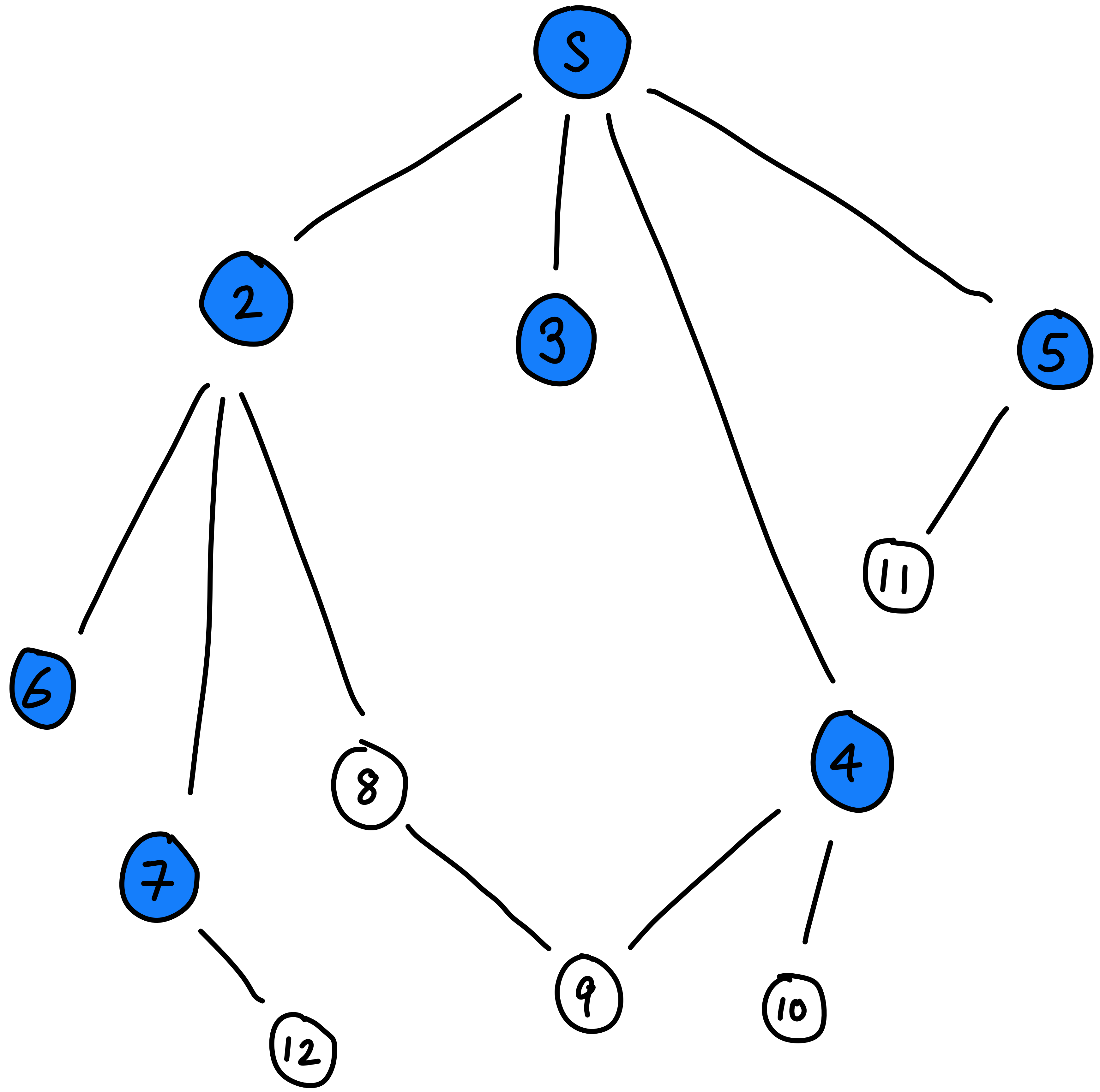


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
8
9
10
11
12

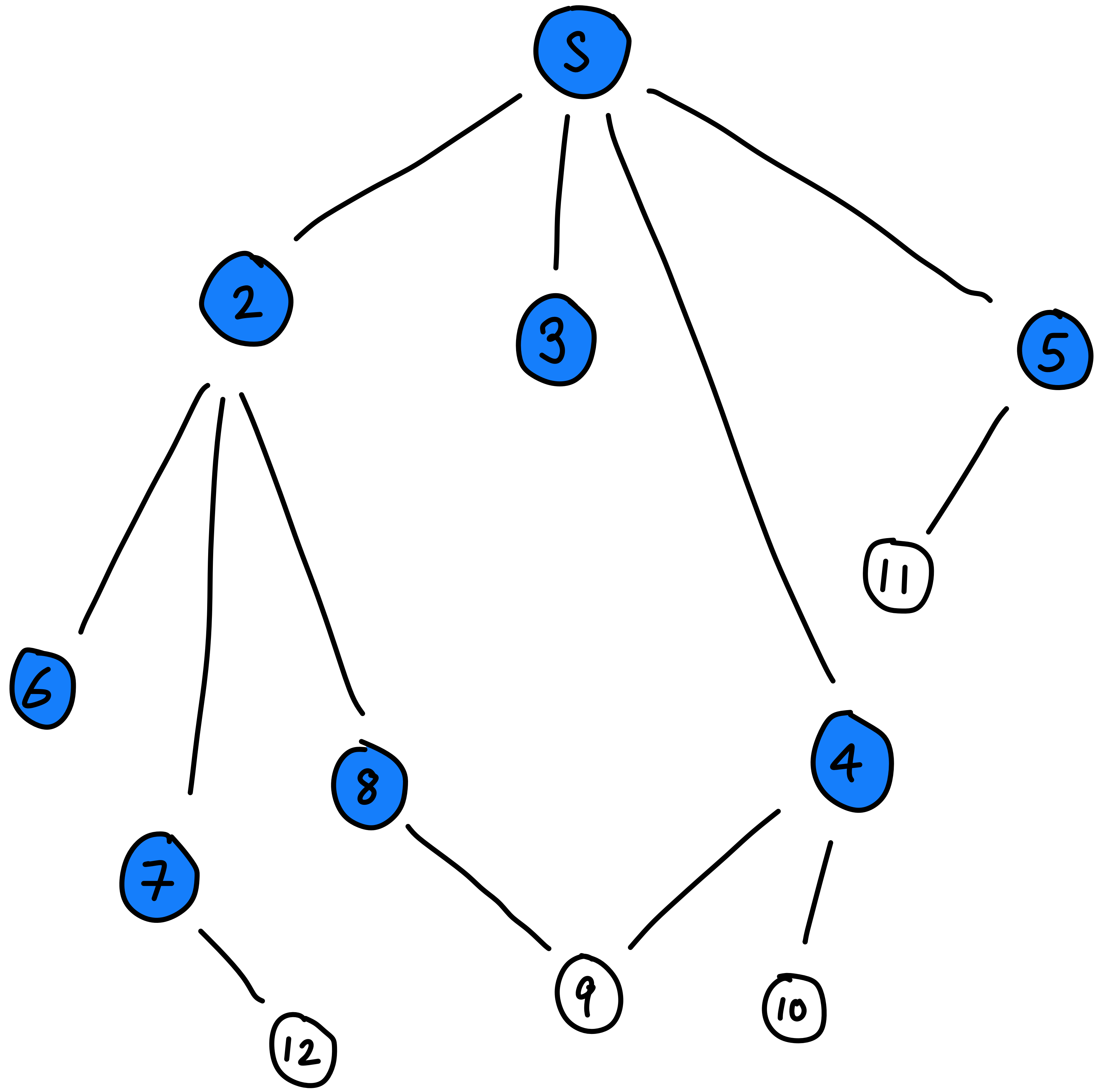


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
9
10
11
12

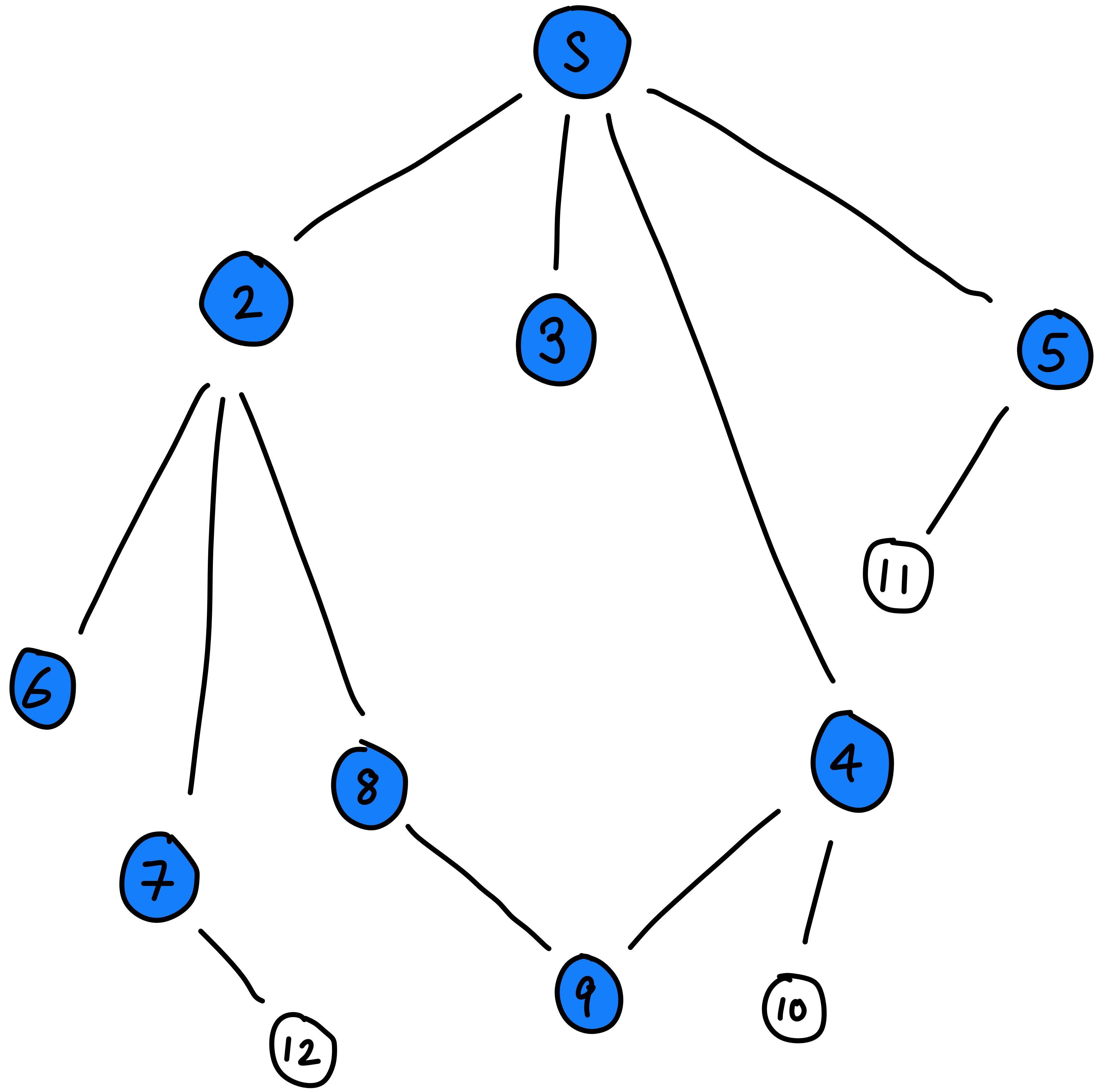


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
10
11
12

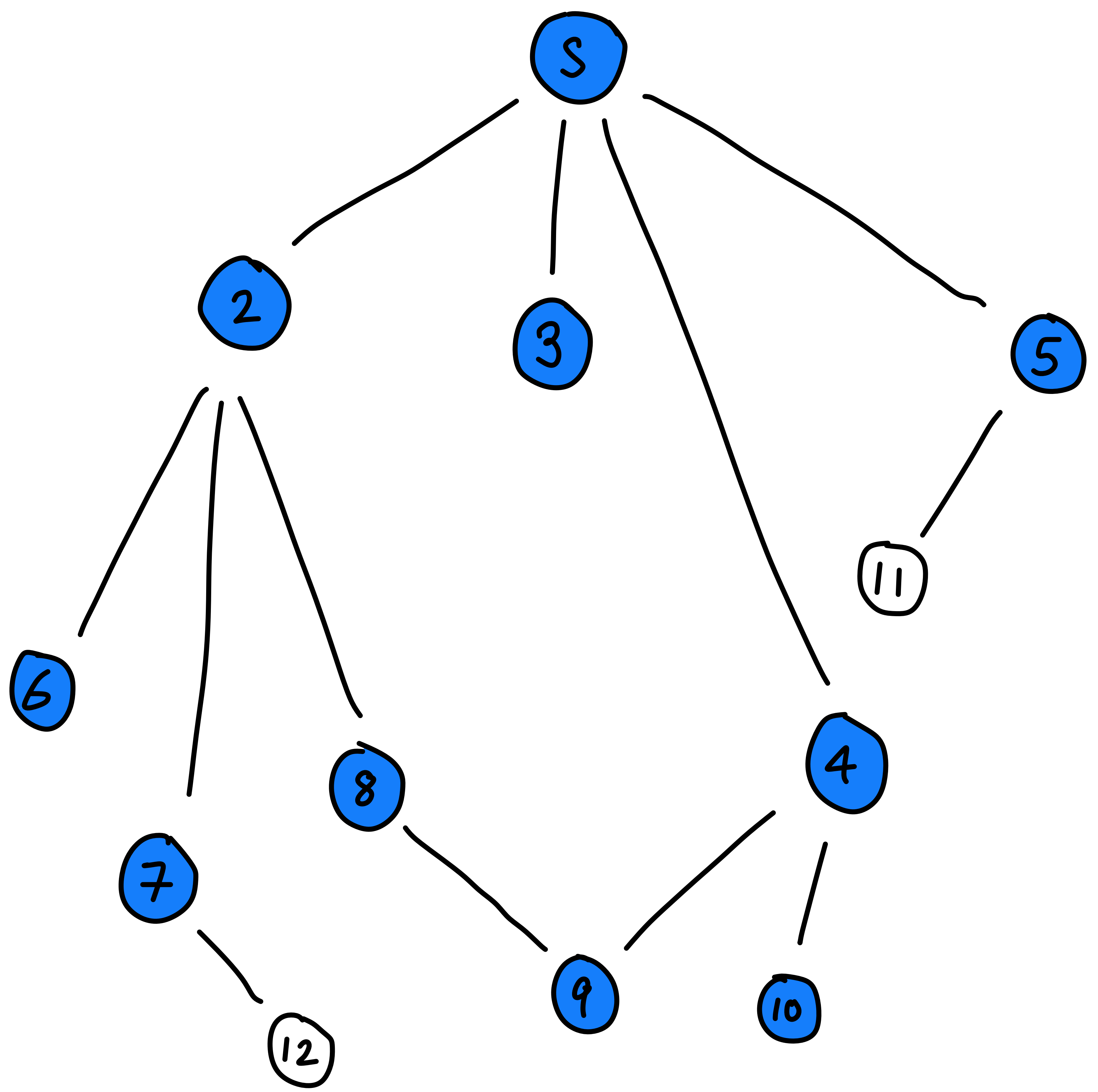


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
11
12

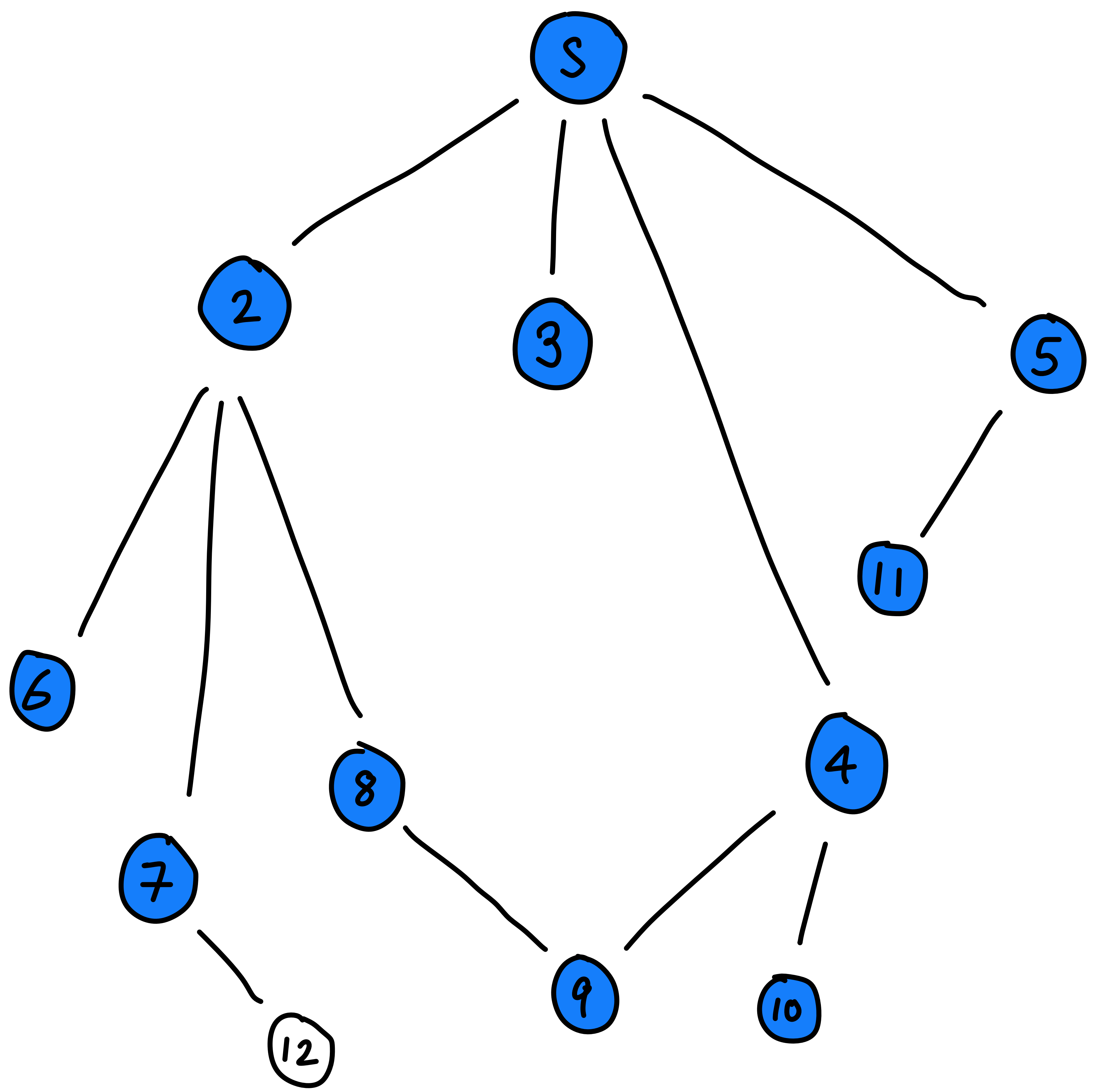


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
~~11~~
12

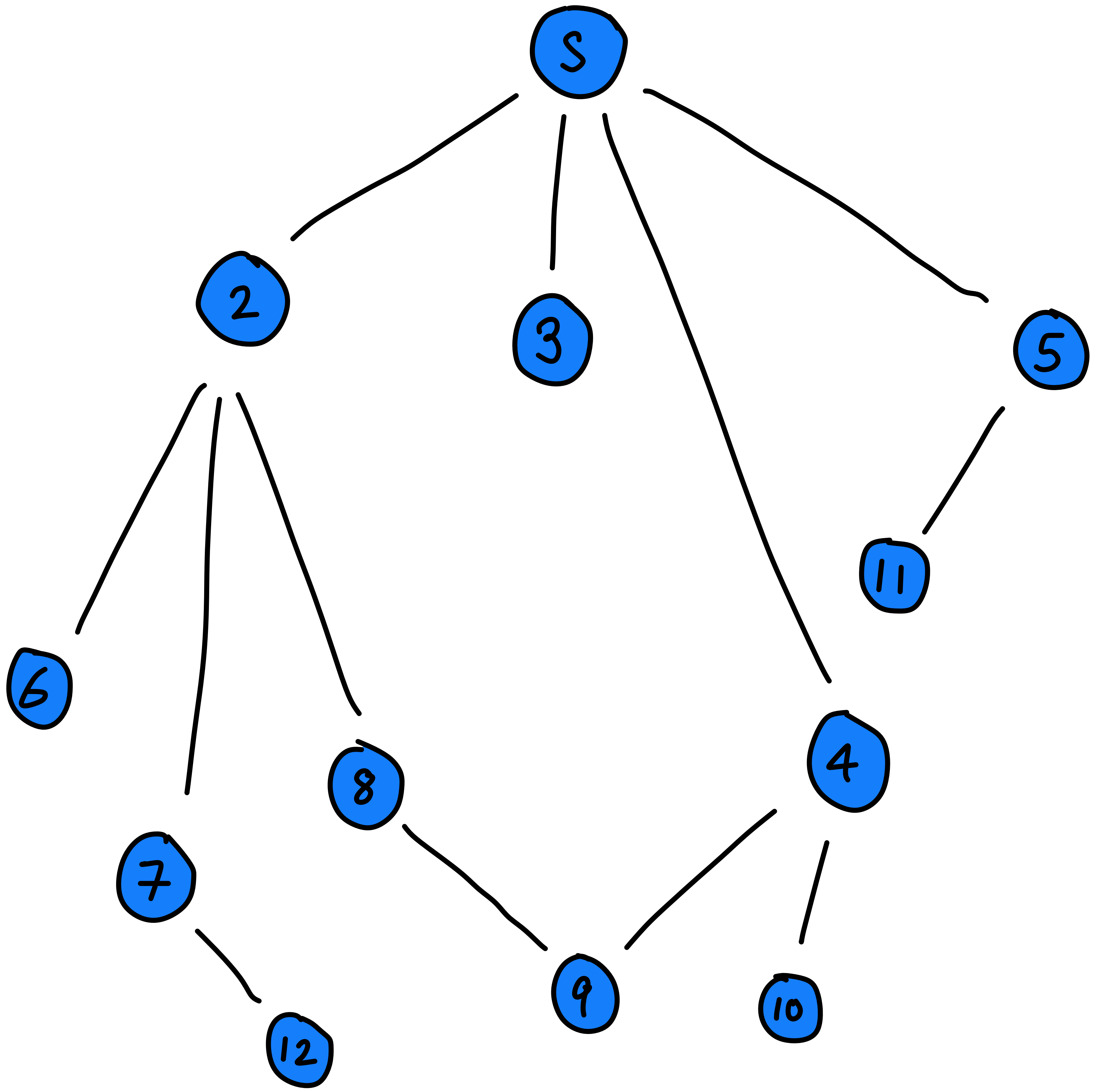


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

queue Q

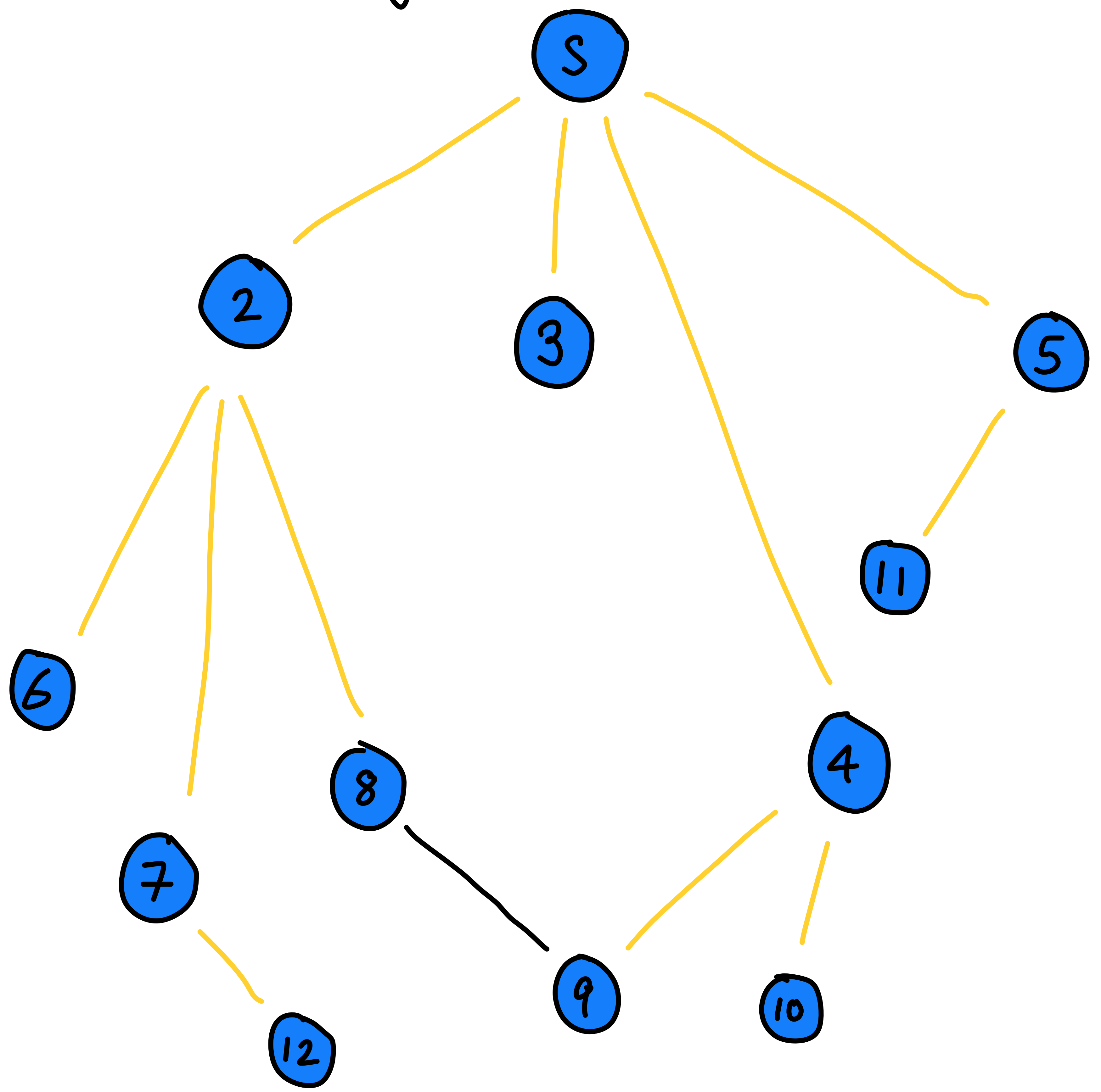
~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
~~11~~
~~12~~



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.

BFS tree generated by tracking which edges are used



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.