# Union-Find and Greedy Algorithms

CSE 101: Design and Analysis of Algorithms

Lecture 8

# CSE 101: Design and analysis of algorithms

- Union-find
  - Reading: Section 5.1
- Greedy algorithms
  - Reading: Kleinberg and Tardos, sections 4.1, 4.2, and 4.3
- Homework 3 is due today, 11:59 PM
- Homework 4 will be assigned today
  - Due Oct 30, 11:59 PM

# Kruskal's algorithm using a DSDS

```
procedure kruskal(G,w)
  Input: undirected connected graph G with edge weights w
  Output: a set of edges X that defines a minimum spanning tree of G
for all v in V                                  |V| * makeset
  makeset(v)
X = { }
Sort the edges in E in increasing order by weight        sort(|E|)
For all edges (u,v) in E until |X| = |V| - 1             2 * |E| * find
  if find(u) ≠ find(v):
    Add edge (u,v) to X
    union(u,v)                                  (|V| - 1) * union
```

# Kruskal's algorithm, DSDS subroutines

- makeset(u)
  - Creates a set with one element, u
- find(u)
  - Finds the set to which u belongs
- union(u,v)
  - Merges the sets containing u and v

- Kruskal's algorithm

  |V| * makeset + 2 * |E| * find + (|V| - 1) * union + sort(|E|)

# DSDS, leader version

- Keep an array leader(u) indexed by element
- In each array position, keep the leader of its set
- makeset(u): leader(u) = u, O(1)
- find(u): return leader(u), O(1)
- union(u,v): For each array position, if it is currently leader(v), then change it to leader(u). O(|V|)

- Kruskal's algorithm

  |V| * makeset + 2 * |E| * find + (|V| - 1) * union + sort(|E|)

  $= |V| * O(1) + 2 * |E| * O(1) + (|V| - 1) * O(|V|) + sort(|E|)$

  $= O(|V|^2)$

# DSDS, directed trees with ranks version

procedure makeset(x)
 $\pi(x) := x$
 rank(x):=0

procedure find(x)
 while $\left(x \neq \pi(x)\right)$
  $x := \pi(x)$
 return x

makeset O(1)
find O(height of tree containing x)
union O(find)

procedure union(x,y)
 rx:=find(x)
 ry:=find(y)
 if rx=ry then return
 if rank(rx)>rank(ry) then
  $\pi(ry) := rx$
 else
  $\pi(rx) := ry$
  if rank(rx)=rank(ry) then
   rank(ry):=rank(rx)+1

# Height of tree

- Any root node of rank k has at least $2^k$ vertices in its tree
- Proof
  - Base Case: a root of rank 0 has 1 vertex
  - Suppose a root of rank k has at least $2^k$ vertices in its tree. Then, a root of rank k+1 can only be made by unioning 2 roots each of rank k. So, a root of rank k+1 must have at least $2^k + 2^k = 2^{k+1}$vertices in its tree.

# Ancestors of rank k

- Any vertex has at most one ancestor of rank k
- Proof
  - Each vertex has one pointer and ranks strictly increase along paths so each element has at most one ancestor of each rank

# Number of vertices of a given rank

- If there are n vertices overall, there can be at most $\frac{n}{2^k}$ vertices of rank k
- Proof
  - Each rank k vertex has at least $2^k$ vertices in its tree so if there are m rank k vertices then there are m$2^k$ vertices overall
- If there are n vertices overall then m$2^k \leq n$ so m$\leq \frac{n}{2^k}$

# Height of tallest tree (i.e., maximum rank)

- The maximum rank is log(n)
- Proof (sort of)
  - How many vertices of rank log(n) can there be?

# Height of tallest tree (i.e., maximum rank)

- The maximum rank is log(n)
- Proof (sort of)
  - How many vertices of rank log(n) can there be?

$$\frac{n}{2^{\log(n)}} = 1$$

anything more is not possible

# DSDS, directed trees with ranks version

- makeset(u)
  - Creates a set with one element, u
- find(u)
  - Finds the set to which u belongs
- union(u,v)
  - Merges the sets containing u and v

makeset O(1)
find O(height of tree containing x)
union O(find)

- Kruskal's algorithm

  |V| makeset + 2 |E| * find + (|V| - 1) union + sort(|E|)

# DSDS, directed trees with ranks version

- makeset(u)
  - Creates a set with one element, u
- find(u)
  - Finds the set to which u belongs
- union(u,v)
  - Merges the sets containing u and v

makeset O(1)
find O(log(|V|))
union O(log(|V|))

- Kruskal's algorithm

  |V| makeset + 2 |E| * find + (|V| - 1) union + sort(|E|)

  $= |V|O(1) + 2|E|O(\log(|V|)) + (|V| - 1)O(\log(|V|)) + |E|\log(|E|)$

  $= O(|V| + |E|\log(|V|) + |V|\log(|V|) + |E|\log(|E|))$

  $= O(|E|\log(|V|))$

# DSDS, directed trees with ranks version

- Kruskal's algorithm

$$O(|E|\log(|V|))$$

makeset O(1)
find O(log(|V|))
union O(log(|V|))

- Can we improve the runtime of find and union?
- Is it worth it (i.e., will it actually improve the runtime of Kruskal's algorithm)?

# Path compression

- We can improve the runtime of find and union by making the height of the trees shorter
- How do we do that?
  - Every time we call find, we do some housekeeping by moving up every vertex

# Path compression

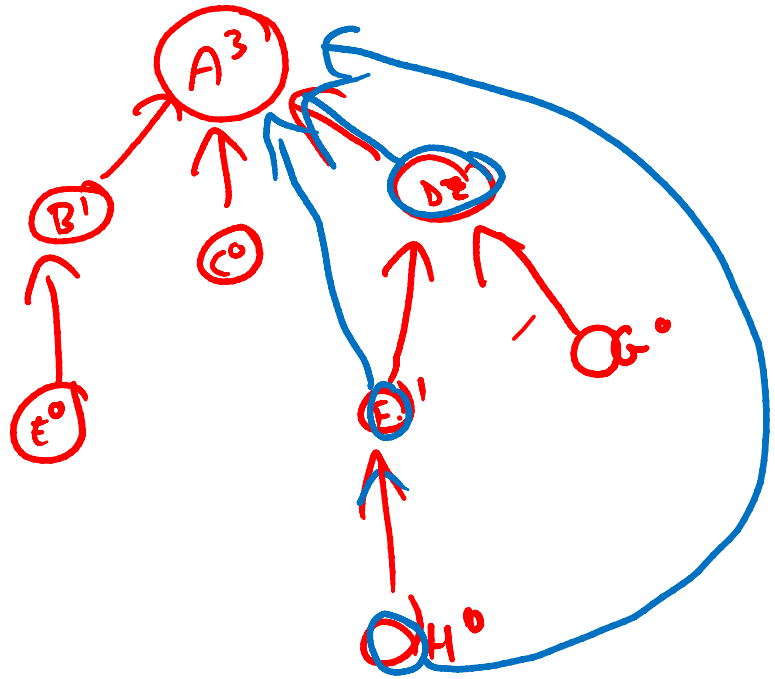
- New find subroutine

procedure find(x)
  if $x \neq \pi(x)$ then
    $\pi(x) := find(\pi(x))$
  return $\pi(x)$

# Path compression

- New find subroutine

procedure find(x)
  if $x \neq \pi(x)$ then
    $\pi(x) := find(\pi(x))$
  return $\pi(x)$




Find (H)

# find, path compression

- Whenever you call find on a vertex v, it points v and all of its ancestors to the root

# find, path compression and ranks

- The ranks do not necessarily represent the height of the graph anymore. Will this cause problems?

# find, path compression and ranks

- The ranks do not necessarily represent the height of the graph anymore. Will this cause problems?
- The following properties still hold
  1. For any x, rank(x)<rank($\pi(x)$)
  2. Any vertex of rank k has at least $2^k$ vertices in its subtree
  3. If there are n vertices overall, there can be at most $n/2^k$ vertices of rank k

# Height of tree with path compression

- How much does this help?
- Recall: the height without path compression is log(n)
- With path compression, the cost of calling find m times has amortized cost of O(m log*(n))

# log*(n)

- log*(n) is the number of logs required to reduce n to a value less than 1
- For example log*(10,000)=4 because

$$\log(10{,}000) = 13.3$$
$$\log\log(10{,}000) = \log(13.3) = 3.7$$
$$\log\log\log(10{,}000) = \log(3.7) = 1.9$$
$$\log\log\log\log(10{,}000) = \log(1.9) = 0.9$$

# log*(n)

- log*(n) is the number of logs required to reduce n to a value less than 1
- For example log*(10,000)=4 because

  $\log(10{,}000) = 13.3$

  $\log\log(10{,}000) = \log(13.3) = 3.7$

  $\log\log\log(10{,}000) = \log(3.7) = 1.9$

  $\log\log\log\log(10{,}000) = \log(1.9) = 0.9$
- It turns out that n must be greater than $2^{65536}$ for log*(n) to be greater than 5

# DSDS, directed trees with ranks version

- makeset(u)
  - Creates a set with one element, u
- find(u)
  - Finds the set to which u belongs
- union(u,v)
  - Merges the sets containing u and v

makeset O(1)
find O(height of tree containing x)
union O(find)

makeset O(1)
find O(log*(|V|))
union O(log*(|V|))

with new find subroutine

- Kruskal's algorithm

  |V| makeset + 2 |E| * find + (|V| - 1) union + sort(|E|)

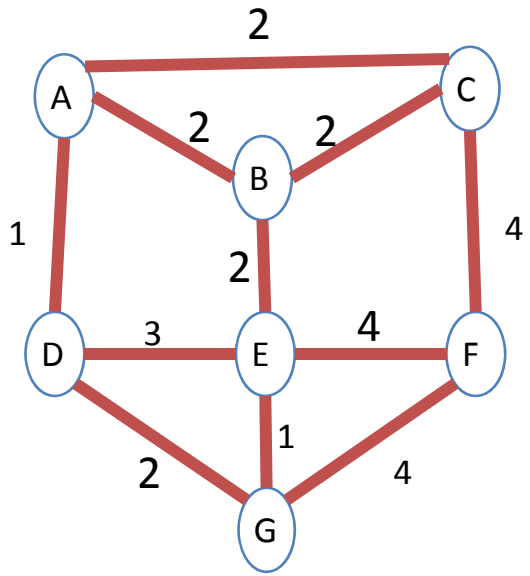
  $= |V|O(1) + |E|O(\log^*(|V|)) + |V|O(\log^*(|V|)) + |E|\log(|E|)$

  $= O((|V| + |E|)\log^*(|V|)) + |E|\log(|E|)$
- If we did not have to sort, then we could perform Kruskal's algorithm in "linear*" time!

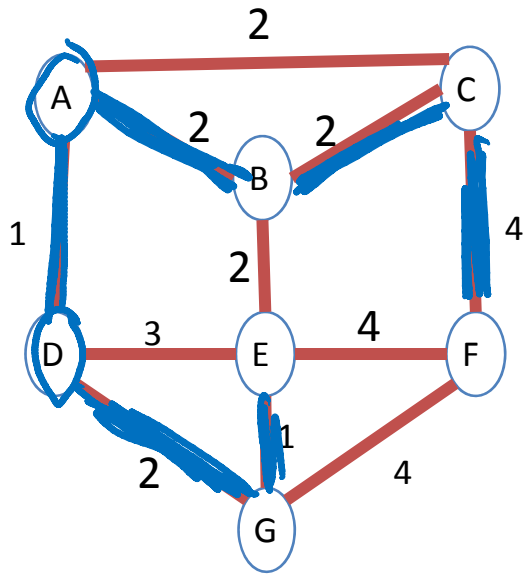
# Prim's algorithm

- Recall Kruskal's algorithm
  - Build a tree by adding the next lightest edge to a forest
- Prim's algoritm
  - Build a tree by adding the next lightest edge connecting the tree to the rest of the vertices

# Example: Prim's algorithm



# Example: Prim's algorithm



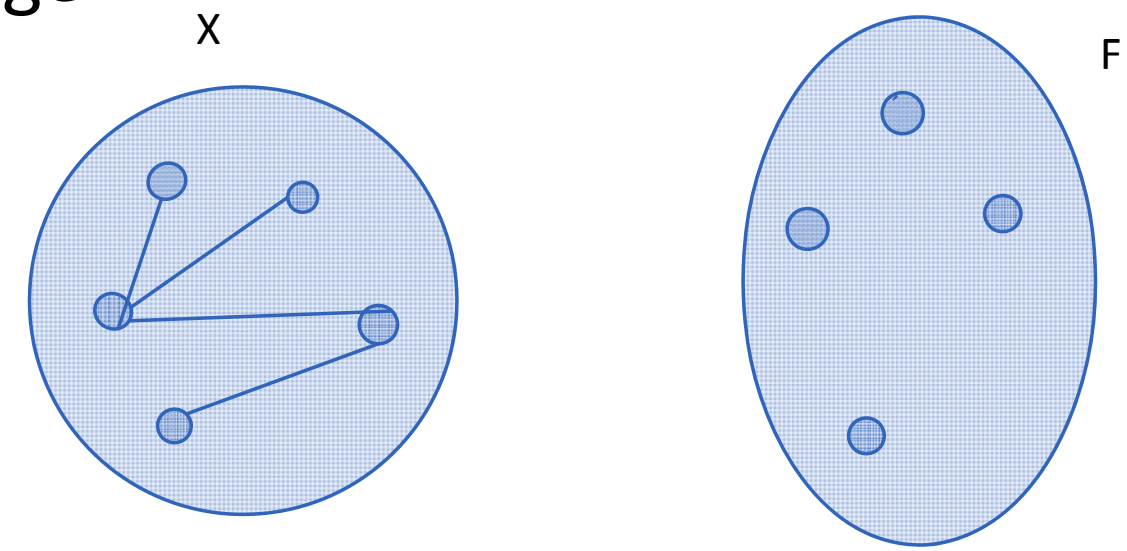
# Dijkstra's algorithm vs Prim's algorithm

- Dijkstra's algorithm
  - Start off with sets X and F
  - Move s from F to X
  - Find the minimum of dist(u)+$\ell(u, v)$ among all edges such that u is in X and v is in F
  - Move v from F to X and repeat

- Prim's algorithm
  - Start off with sets X and F
  - move s from F to X
  - Find the minimum of $w(u, v)$ among all edges such that u is in X and v is in F
  - Move v from F to X and repeat

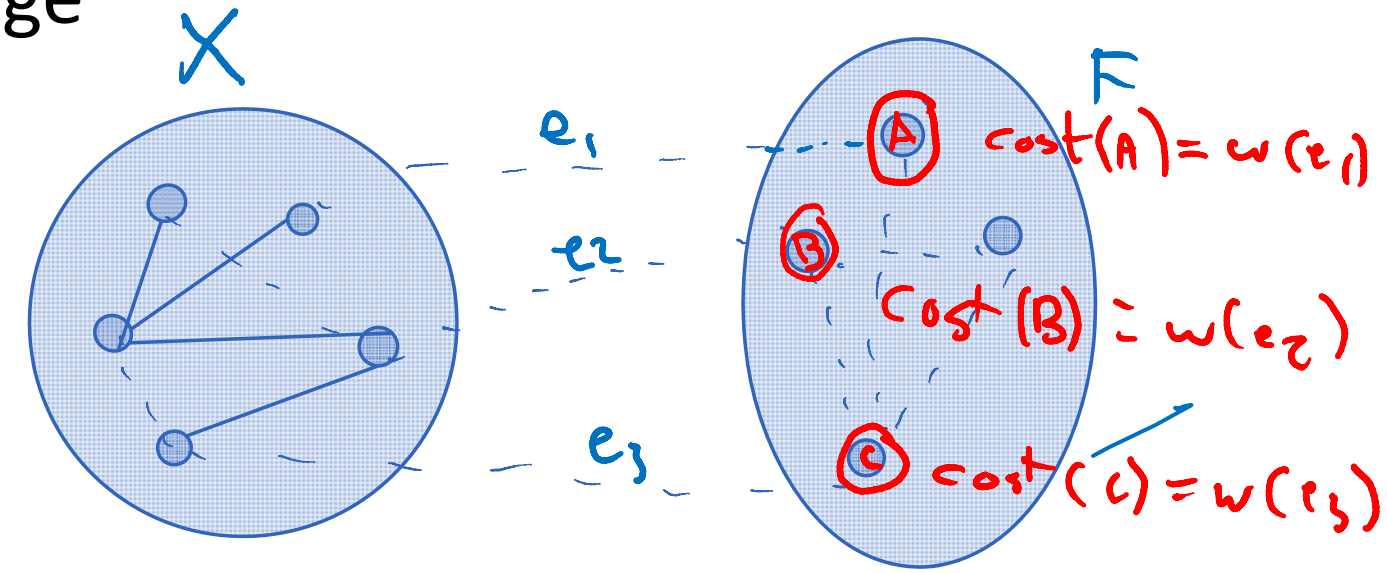
# Prim's algorithm

- On each iteration, the subtree grows by one edge



# Prim's algorithm

- On each iteration, the subtree grows by one edge



# Prim's algorithm

- Prim's algorithm is just like Dijkstra's algorithm, where we are using the value cost(v) instead of dist(v)
- We can use the same algorithm changing dist to cost and we can use the same data structures

# Prim's algorithm

procedure prim(G,w)
Input: A connected undirected graph G=(V,E) with edge weights w(e).
Output: A minimum spanning tree defined by the array prev()

for all $u \in V$:
  cost(u)=infinity
  prev(u)=nil
pick any initial vertex u
cost(u)=0
H=makequeue(V)
while H is not empty
  v=deletemin(H)
  for each $\{v, z\} \in E$:
    if cost(z)>w(v,z) then
      cost(z)=w(v,z)
      prev(z)=v

Same runtime as Dijkstra's algorithm

# GREEDY ALGORITHMS

# Optimization problems

- In general, when you try to solve a problem, you are trying to find the best solution from among a large space of possibilities
- The format for an optimization problem is
  - Instance: what does the input look like?
  - Solution format: what does an output look like?
  - Constraints: what properties must a solution have?
  - Objective function: what makes a solution better or worse?

# Optimization examples

- Shortest path
  - Instance: directed graph G with positive edge weights $\ell(e)$, vertices s,t
  - Solution format: list of edges $e_1, .. e_k$
  - Constraint: must form a path from s to t
  - Objective: minimize $\sum \ell(e_i)$

# Optimization examples

- Max bandwidth path
  - Instance: directed graph G with positive edge weights $\ell(e)$, vertices s,t
  - Solution format: list of edges $e_1, .. e_k$
  - Constraint: must form a path p from s to t
  - Objective: maximize $\min_{e \in p} w(e)$

# Global search vs local search

- In general, there will be exponentially many possible solutions
- The obvious algorithm is to try them all and take the best, but this is usually prohibitively slow. Sometimes, this is unavoidable (unless P=NP).
- A good way to make progress towards an efficient algorithm is to break the massive global search for a solution into a series of simpler local searches for part of the solution (e.g., Which edge do we take first? Then second? …)
- If you cannot tell which local choice is best, you may still have to use **exhaustive search** to try out all combinations of local decisions and find the optimal one

# The greedy method

- In some cases (certainly not all) there is sufficient structure that allows you to reach the correct solution by just picking the straightforward “best” decision at each stage
- This is called the greedy method

# The greedy method

- In some cases (certainly not all) there is sufficient structure that allows you to reach the correct solution by just picking the straightforward “best” decision at each stage
- This is called the greedy method
  - It does not always work
  - Just as in life, acting in one’s immediate best interest is not always the best longer-term strategy

# Other uses of local decisions

- Many of the other techniques we'll study are also based on breaking up global search into local decisions
  - Backtracking
  - Dynamic programming
  - Hill-climbing
  - (Stochastic search heuristics)

# Cookies

- Suppose you are the cookie monster and you have a 6x6 sheet of freshly baked cookies in front of you. The cookies are all chocolate chip cookies but they may have different sizes.
- If you are only allowed to take six cookies, how can you maximize your total cookie intake?
- Devise an algorithm to do this

# Problem specification

- Cookies
  - Instance: sizes of cookies on 6x6 sheet
  - Solution format: set of 6 cookies
  - Constraint: none
  - Objective: maximize sum of sizes

# Cookies

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

1. What is an algorithm you could use to select the *best* option?

(The best option means the sum of all the cookie sizes is the highest possible)

# Cookies

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

1. What is an algorithm you could use to select the *best* option?

(The best option means the sum of all the cookie sizes is the highest possible)

99+97+94+92+88+85=555

# Problem specification

- Cookies, one per row
  - Instance: sizes of cookies on 6x6 sheet
  - Solution format: set of 6 cookies
  - Constraint: one cookie per row
  - Objective: maximize sum of sizes

# Cookies, one per row

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

2. What is an algorithm you could use to select the *best* option if you can only select one cookie from each row?

# Cookies, one per row

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

2. What is an algorithm you could use to select the *best* option if you can only select one cookie from each row?

76+80+99+68+85+73=481

# Problem specification

- Cookies, cannot share same row or column
  - Instance: sizes of cookies on 6x6 sheet
  - Solution format: set of 6 cookies
  - Constraint: no two cookies share the same row or column
  - Objective: maximize sum of sizes

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

99+81+74+60+50+40=404

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

99+81+74+60+50+40=404

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

99+81+74+60+50+40=404

99+81+72+69+47+46=414

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

3. What is an algorithm you could use to select the *best* option if you can't select 2 cookies from the same row or column?

99+81+74+60+50+40=404

99+81+72+69+47+46=414

92+78+75+73+72+68=458

# The greedy method

- As you have seen, the greedy method does not always work. Because of this, in order to use the greedy method, we must prove the correctness of the algorithm.
  - Or else, we must present a counterexample to show that a particular greedy method will not work

# The greedy method

- As you have seen, the greedy method does not always work. Because of this, in order to use the greedy method, we must prove the correctness of the algorithm.
  - Or else, we must present a counterexample to show that a particular greedy method will not work
- Warning: Furthermore, for a single problem, there may be more than one potential greedy strategy (i.e., more than one way to choose the "best" possible choice at each step). The problem may be solved in one way but not the other.

# Immediate benefit vs opportunity costs

- Immediate benefit:  how does the choice we are making now contribute to the objective function?
- Opportunity costs:  How does the choice we are making now restrict future choices?
- The greedy method (usually) takes the best immediate benefit and ignore opportunity costs
- The greedy method is optimal: best immediate benefits outweigh opportunity costs

# Cookies, one per row

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

99

Immediate benefit

82, 97, 94, 88, 92: Opportunity costs

(Since we can have at most one of these:97)

Immediate benefit > Opportunity cost

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

99

Immediate benefit

Opportunity costs

(Lose out on one in row, one in column: 97+85)

Immediate benefit < Opportunity cost

# Cookies, cannot share same row or column

| | | | | | |
|---|---|---|---|---|---|
| 56 | 76 | 69 | 60 | 75 | 51 |
| 61 | 77 | 74 | 72 | 80 | 58 |
| 82 | 97 | 94 | 88 | 99 | 92 |
| 47 | 68 | 59 | 52 | 65 | 40 |
| 78 | 81 | 79 | 71 | 85 | 62 |
| 50 | 67 | 73 | 57 | 70 | 46 |

Immediate benefit < Opportunity cost

99

Immediate benefit

Opportunity costs

(Lose out on one in row, one in column: 97+85)

# Event scheduling

- Suppose you are running a cookie conference and you have a collection of events (or talks) that each have a start time and an end time
- Problem: you only have one conference room!
- Your goal is to schedule the most events possible that day such that no two events overlap

# Problem specification

- Event scheduling
  - Instance: list of events $E_1, \ldots, E_n$; $E_i = (s_i, f_i)$
  - Solution format: subset of $E_1, \ldots, E_n$
  - Constraint: no two events overlap
  - Objective: maximize size of subset

# Event scheduling

- Your goal is to schedule the most events possible that day such that no two events overlap
- Brute Force: Say that T is the set of events and |T|=n
- Check all possibilities
  - How would we do that?

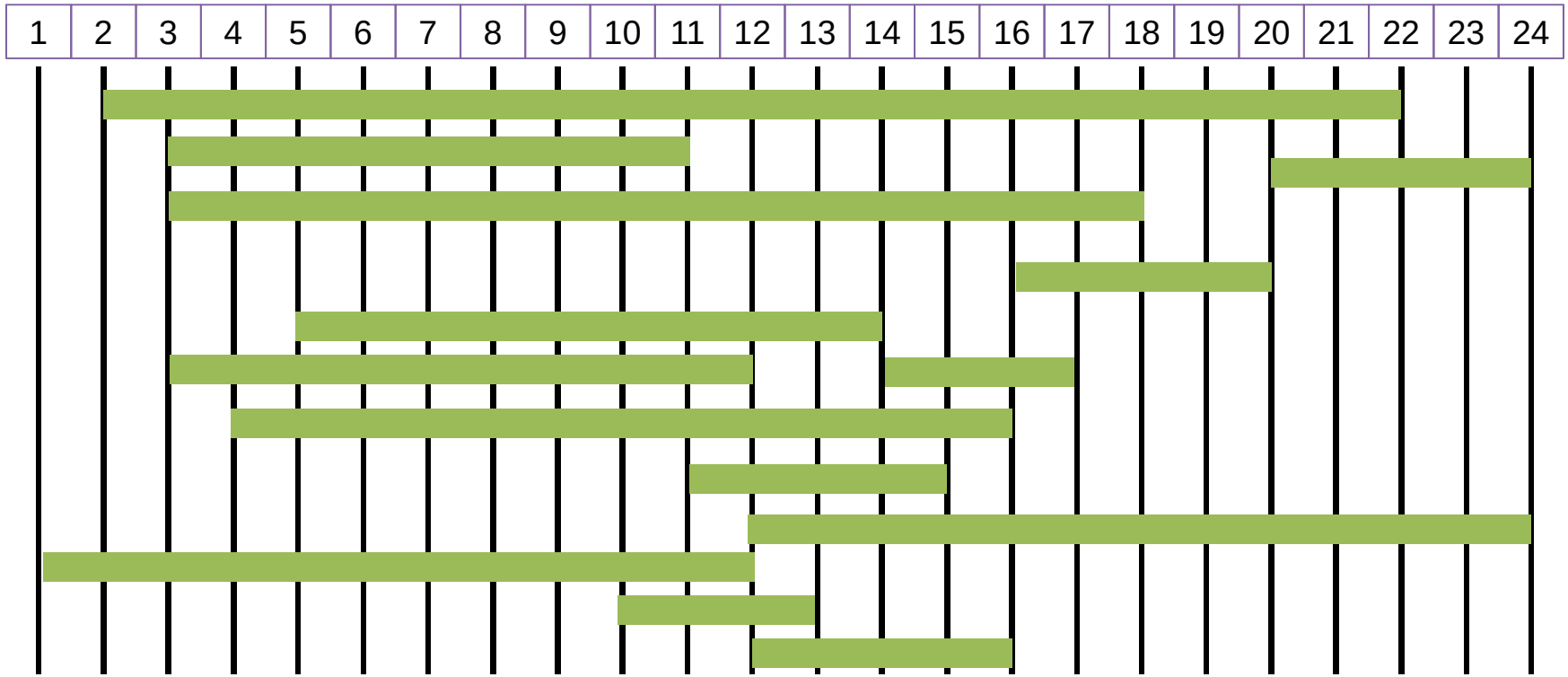
# Event scheduling

- Your goal is to schedule the most events possible that day such that no two events overlap
- Brute Force: Say that T is the set of events and |T|=n
- Check all possibilities
  - How would we do that?
- Go through all subsets of T. Check if it is a valid schedule (i.e., no conflicts) and count the number of events.
- Take the maximum out of all valid schedules
- How many subsets of T are there?

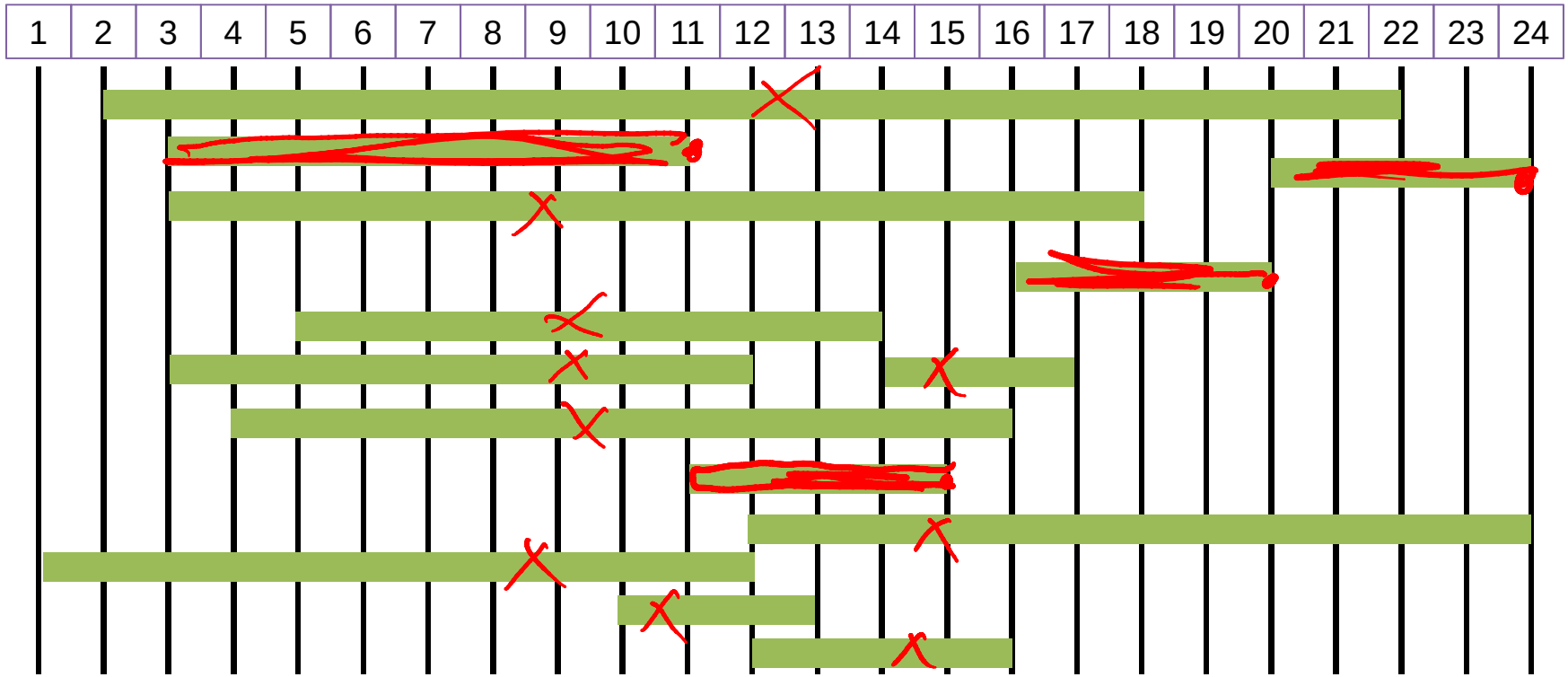
# Event scheduling

- Your goal is to schedule the most events possible that day such that no two events overlap
- Brute Force: Say that T is the set of events and $|T|=n$
- Check all possibilities
  - How would we do that?
- Go through all subsets of T. Check if it is a valid schedule (i.e., no conflicts) and count the number of events.
- Take the maximum out of all valid schedules
- How many subsets of T are there? $2^n$

# Event scheduling



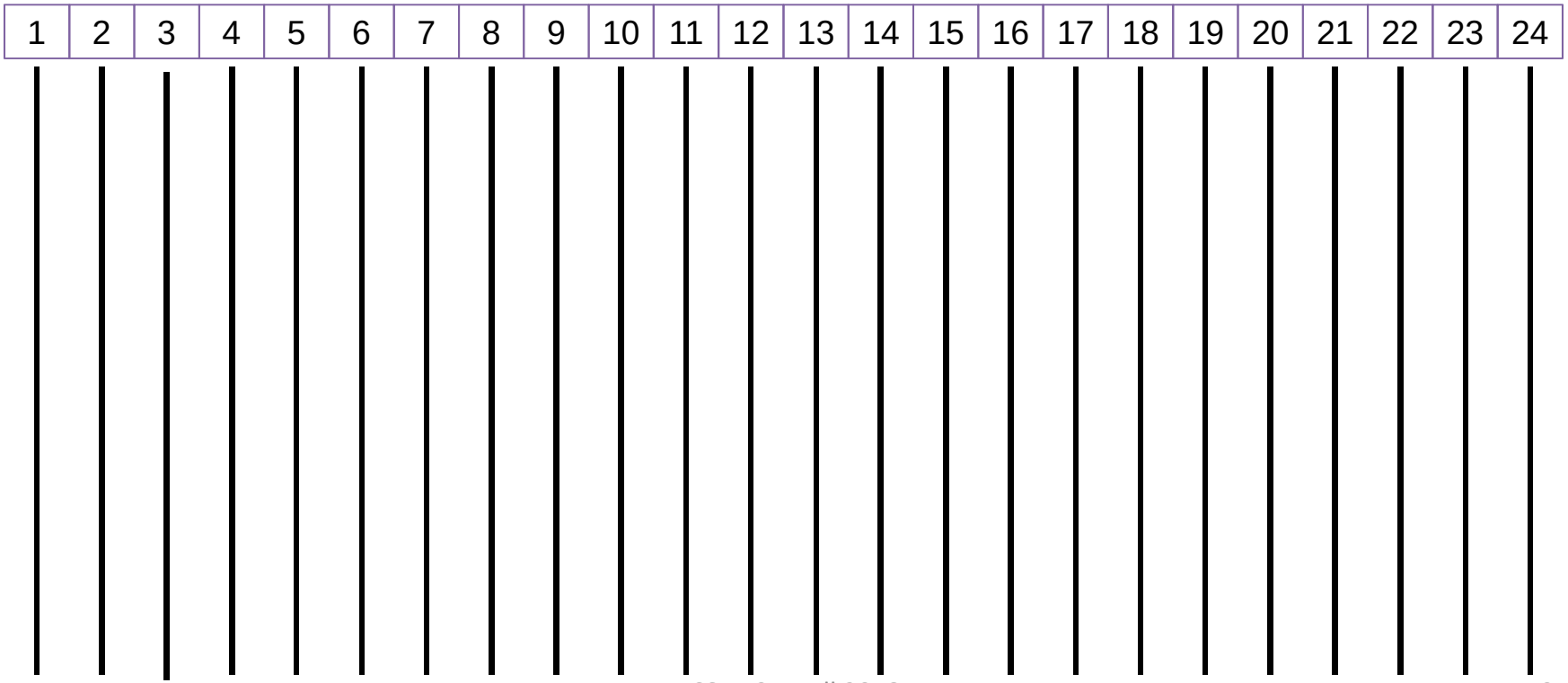
# Event scheduling



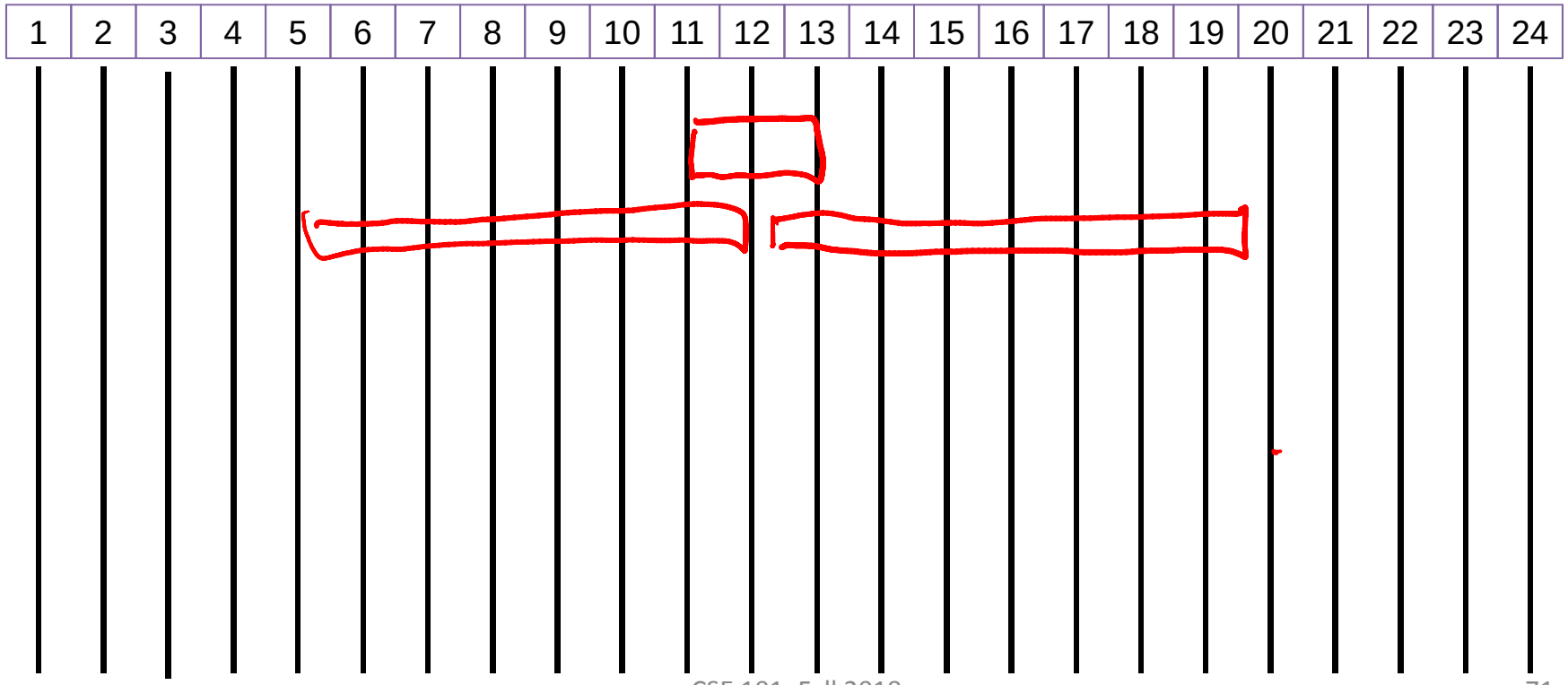
# Event scheduling

- Your goal is to schedule the most events possible that day such that no two events overlap
- Exponential is too slow. Let's try some greedy strategies:
  - Shortest duration
  - Earliest start time
  - Fewest conflicts (take away most conflicts)
  - Earliest end time

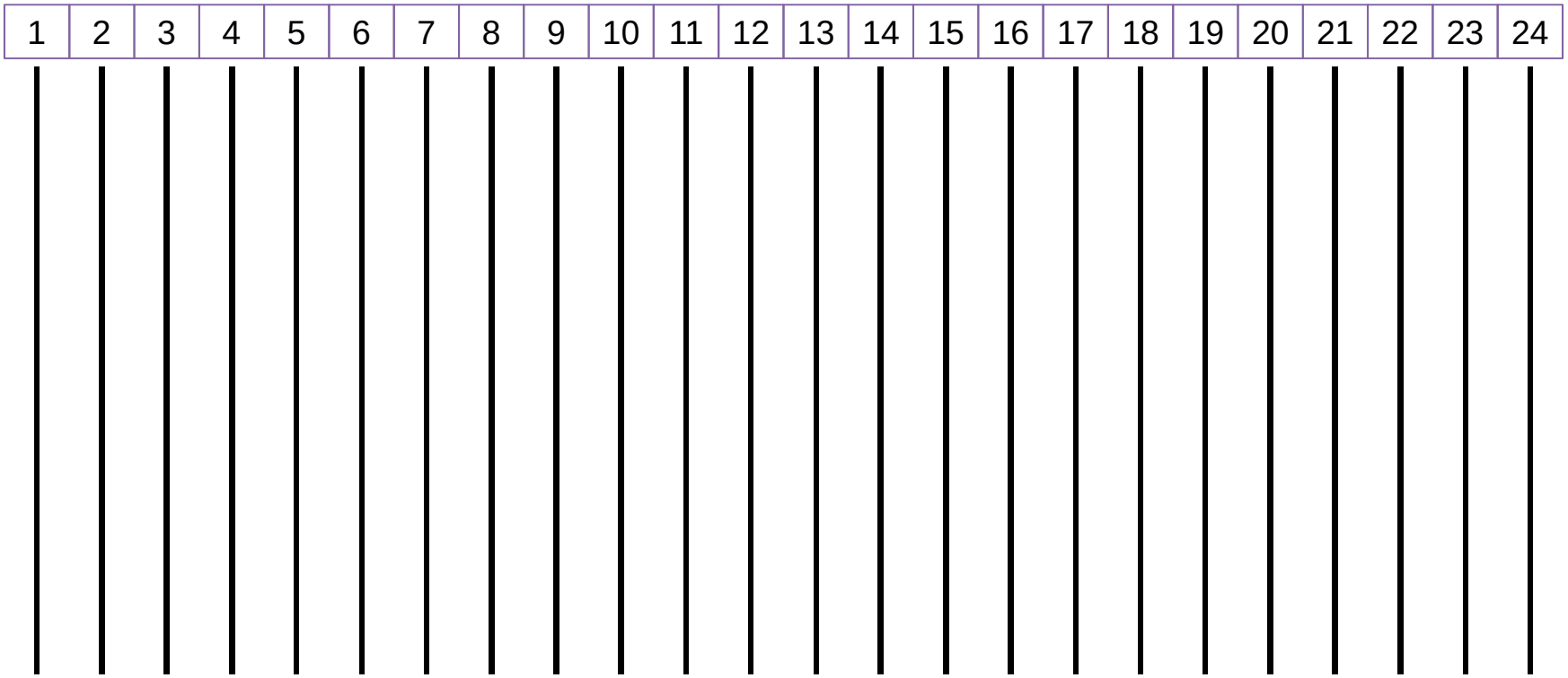
# Shortest duration



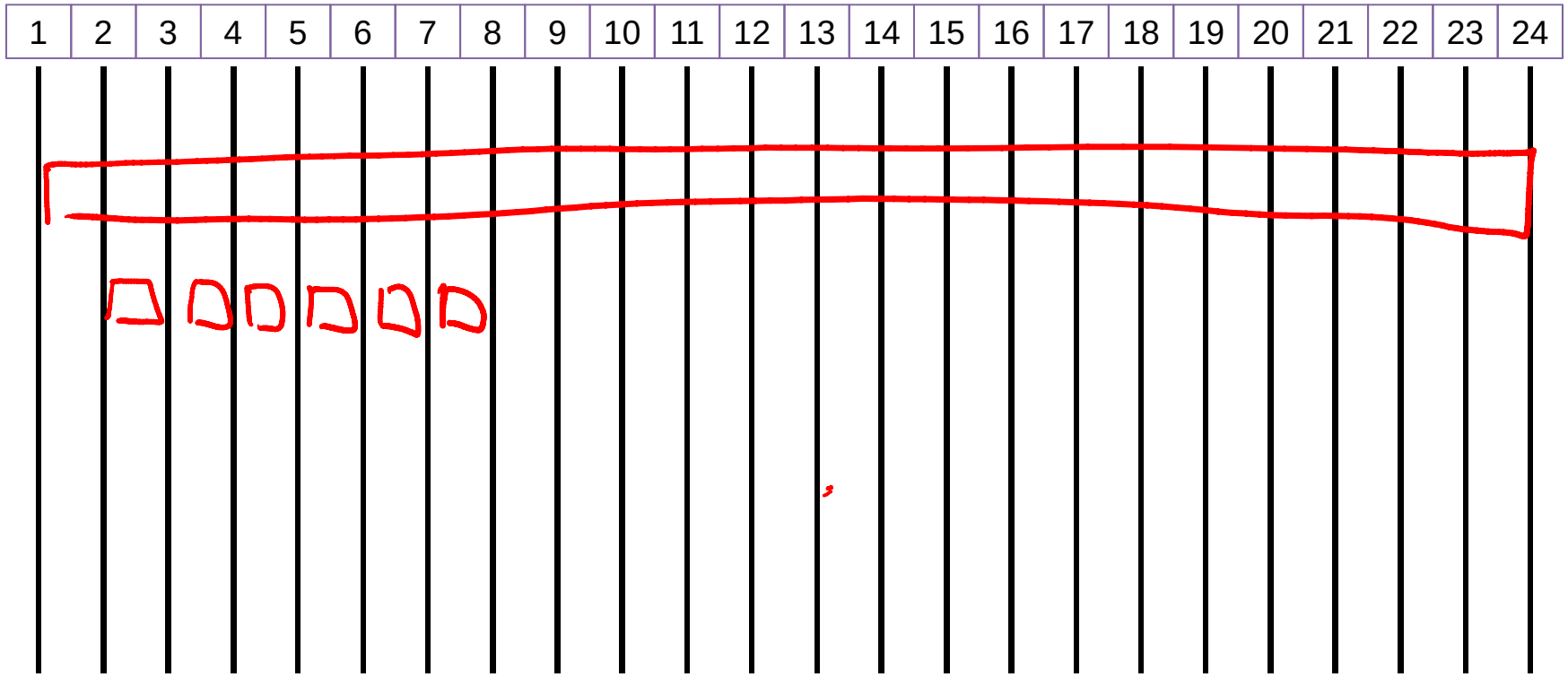
# Shortest duration



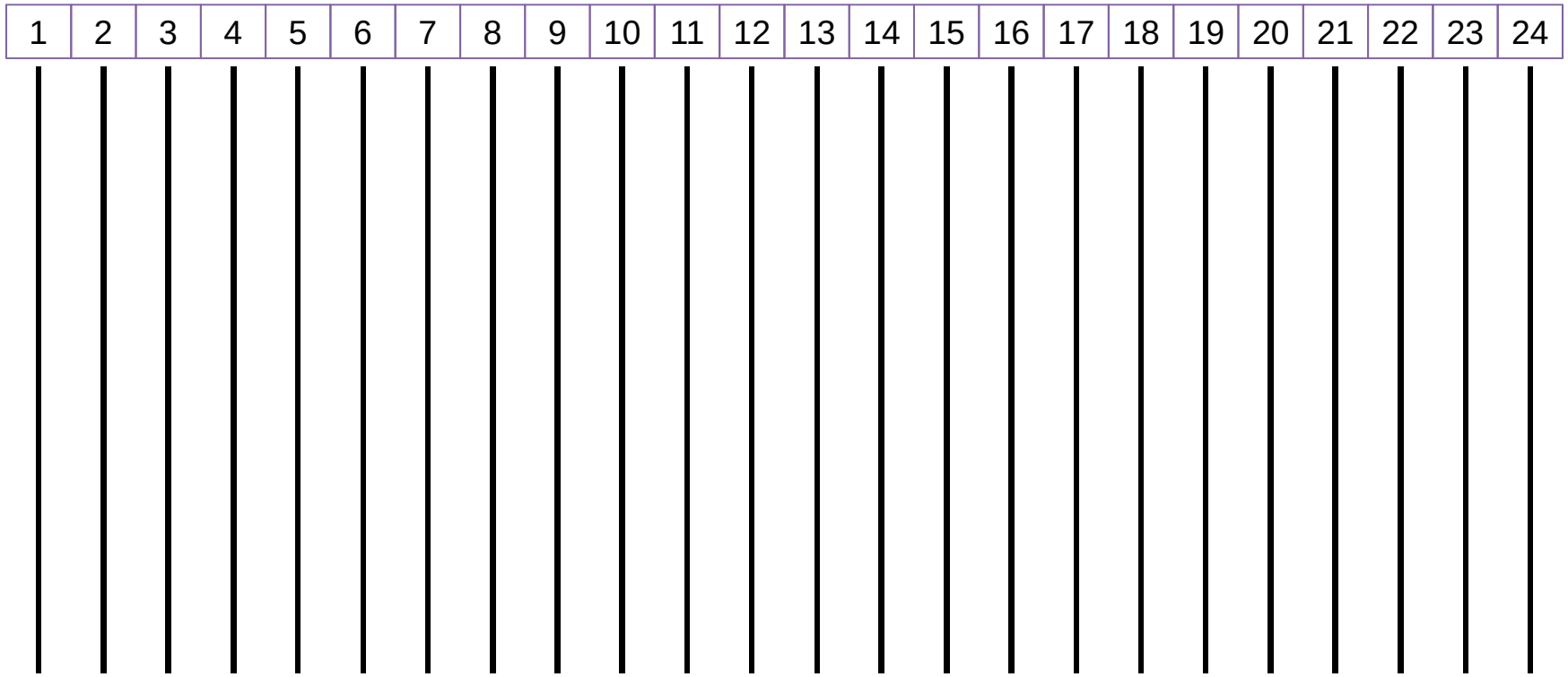
# Earliest start time



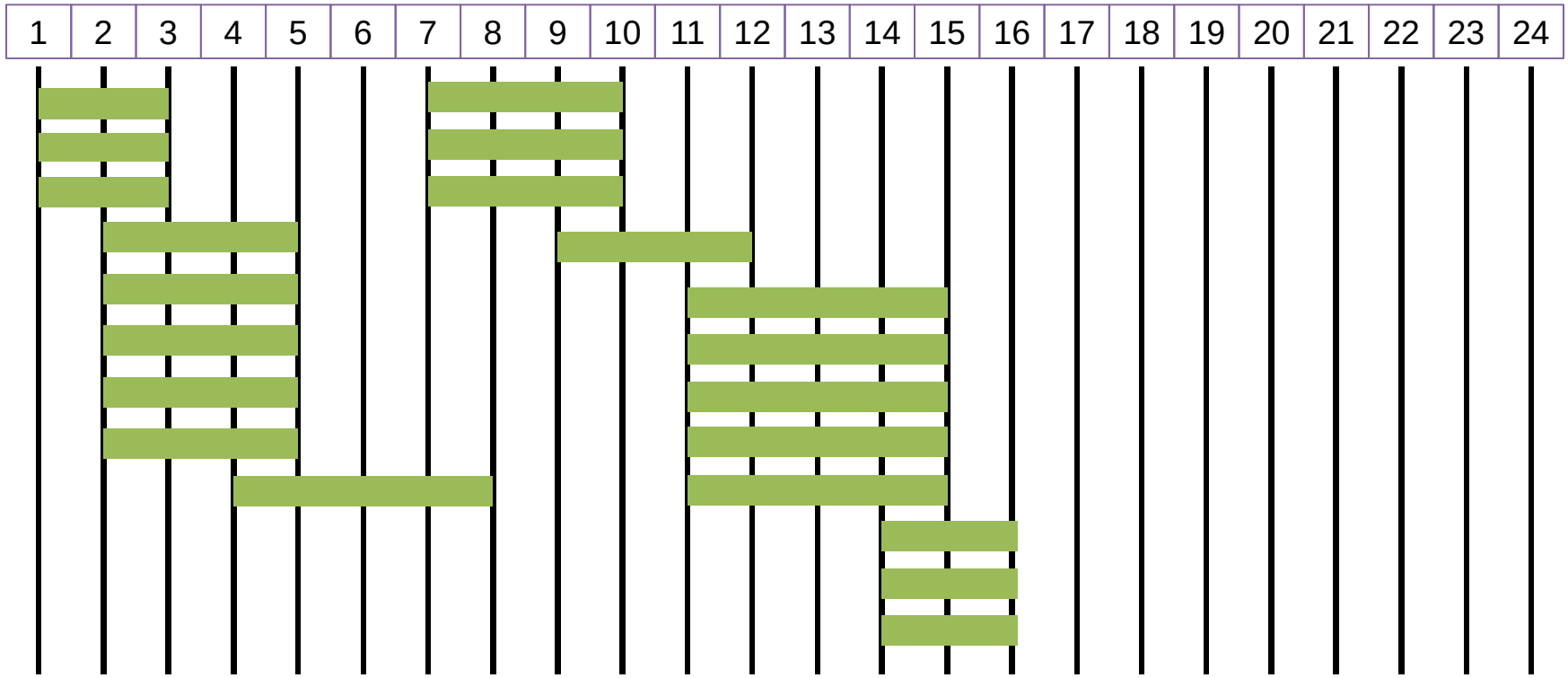
# Earliest start time



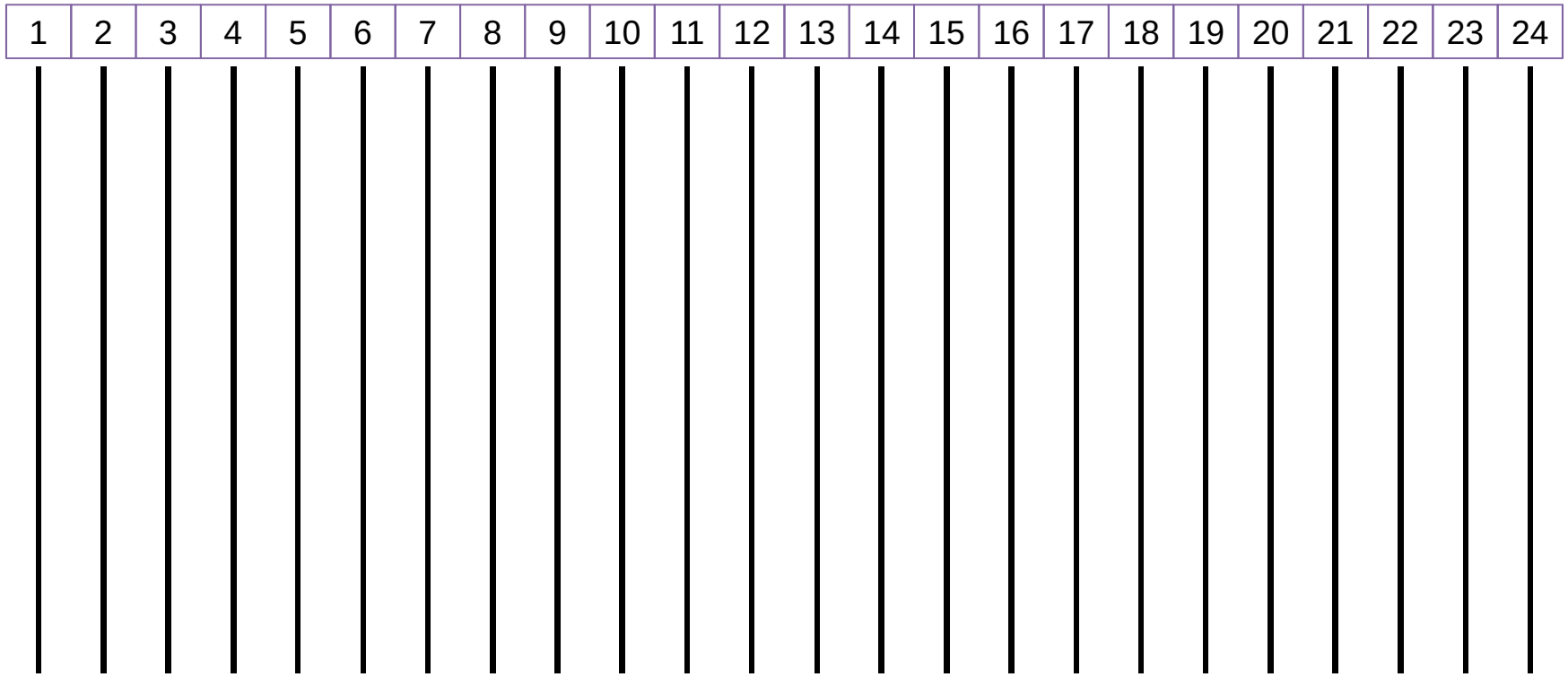
# Fewest conflicts



# Counterexample for fewest conflicts



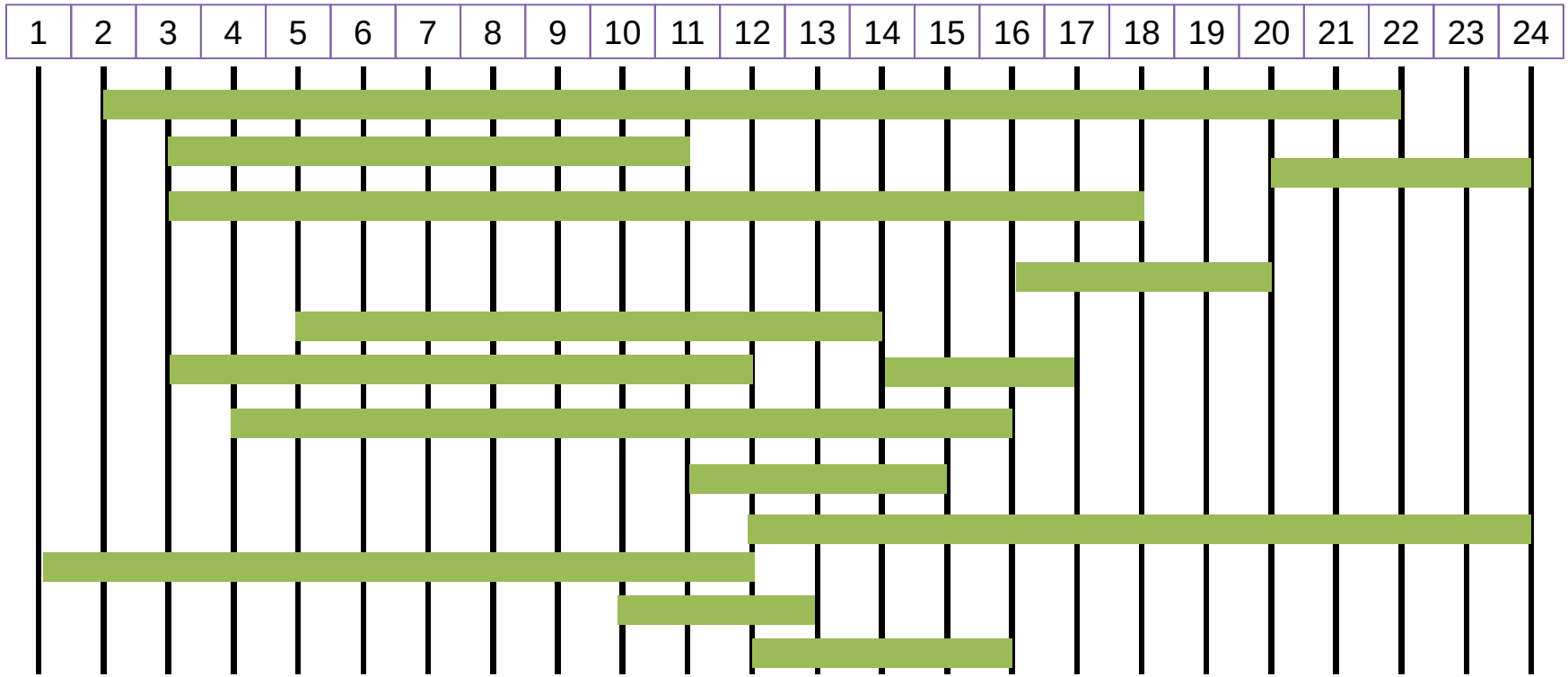
# Earliest end time



# Event scheduling

- Your goal is to schedule the most events possible that day such that no two events overlap
- Exponential is too slow. Let's try some greedy strategies:
  - ~~Shortest duration~~
  - ~~Earliest start time~~
  - ~~Fewest conflicts~~
  - Earliest end time (we cannot find a counterexample!)

# Event scheduling



# Problem specification

- Design an algorithm that uses the greedy choice of picking the next available event with the earliest end time
  - Instance: $n$ events each with a start and end time
  - Solution format: list of events
  - Constraints: events cannot overlap
  - Objective: Maximize the number of events
- Get into groups and come up with an efficient implementation

# Event scheduling implementation

Initialize a queue $S$
Sort the intervals by end time
Put the first event $E_1$ in $S$
Set $F = f_1$
For $i = 2 \ldots n$:
  If $s_i \geq F$:
    enqueue$(E_i, S)$
    $F = f_i$
Return $S$

# Proving optimization algorithms correct

- Remember what we need to show
  - Let I be any instance of our problem, GS the greedy algorithm's solution, and OS any other solution
  - If minimization: Cost (OS) ≥ Cost(GS)
  - If maximization: Value(GS) ≥ Value (OS)

# Next lecture

- Greedy algorithms
  - Reading: Kleinberg and Tardos, sections 4.1, 4.2, and 4.3