Randomized Algorithms

Ioannis Caragiannis (this time) and Kasper Green Larsen

Practical issues

- When: Tuesdays, 8-11am
- Where: 5510-104 Lille Auditorium
- 3 projects in groups of three
- Oral exam

Grading rules:

- To pass the course, you need to pass the projects and the oral exam
- The projects can affect your grade in the oral exam by one point (up or down)

This lecture

- Randomization: assumptions, characteristics, benefits, and costs
- Examples of randomized algorithms
- Quicksort
- Randomized Quicksort and its analysis

Randomization

The elephant in the room

- Randomized algorithms use **random coins**, **dice**, **card shuffling**, etc

- For example, the code of a randomized algorithm implementation will typically have a line like this:
- `if (coin_toss() == HEADS) {…}`

Usual assumptions

Basic operation:

- Access to **fair coins** (Pr[HEADS]=Pr[TAILS]=1/2)

More complicated operations:

- **Random selection** among a finite set of items
- Access to a **random permutation** of elements
- Selection of a **random point** in the interval [0,1]
- Selection from a finite or infinite set according to a **non-uniform probability distribution**

Note: there are important **implementation issues** that we most of the time ignore

Main characteristics

Deterministic algorithms: performs the very **same steps** in **any execution** on the **same input**

Randomized algorithms do not!

- They may produce **different outputs** in different executions
- Their **running time** may not always the same
- In other words, their output, their running time, the amount of space they use are **random variables**

With the **analysis of randomized algorithms**, our aim is to understand these random variables

Randomized algorithms: Why do we want them?

- Sometimes randomization is absolutely **necessary**
- They are usually **simple**
- Work **well on average** or **with high probability**
- Sometimes, the give insights to the design of better deterministc algorithms (**derandomization**)

Examples of randomized algorithms

Example: Contention resolution

- Access to a communication network (e.g., ethernet)
- Whenever two nodes try to submit messages simultaneously, a **collision** will happen and both messages should be retransmitted
- **Protocols** make sure that messages will be sent correctly

Example: Contention resolution

- Access to a communication network (e.g., ethernet)
- Whenever two nodes try to submit messages simultaneously, a **collision** will happen and both messages should be retransmitted
- **Protocols** make sure that messages will be sent correctly
- Idea: **be patient**; if transmission fails, retry after 15ns
- If a collision happens once, it will happen **again and again** ☹

Example: Contention resolution

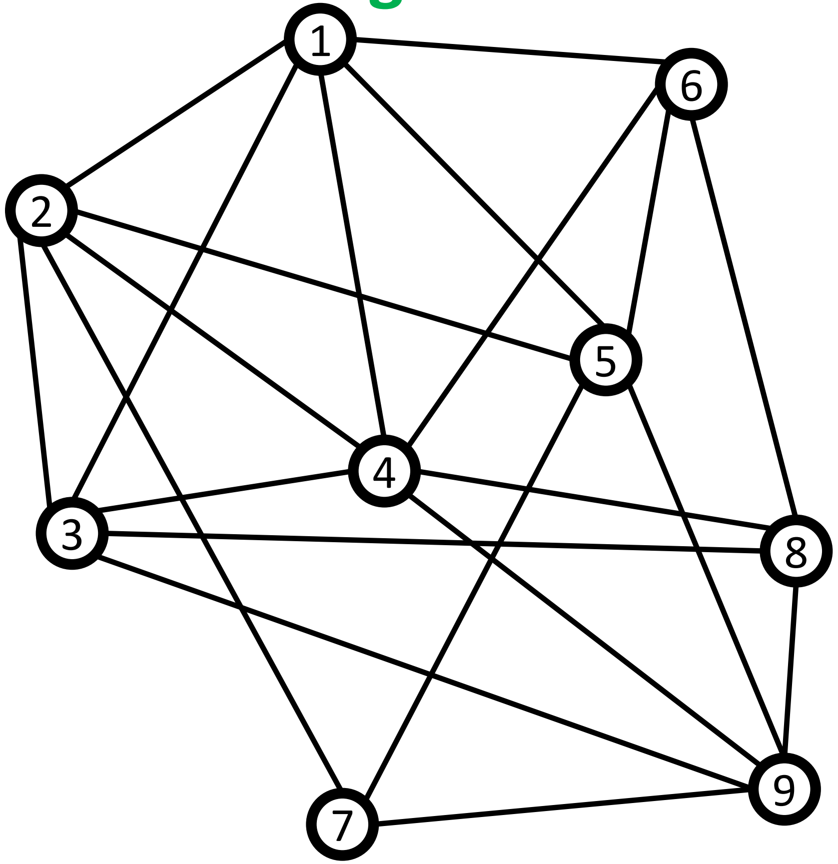
- Access to a communication network (e.g., ethernet)
- Whenever two nodes try to submit messages simultaneously, a **collision** will happen and both messages should be retransmitted
- **Protocols** make sure that messages will be sent correctly
- Idea: **be patient**; if transmission fails, retry after 15ns
- If a collision happens once, it will happen **again and again** ☹
- Idea: If transmission fails, **wait for some random time and retry**
- Most probably, retransmissions will take place in **different time slots** ☺
- Important parameters: **how much to wait?**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

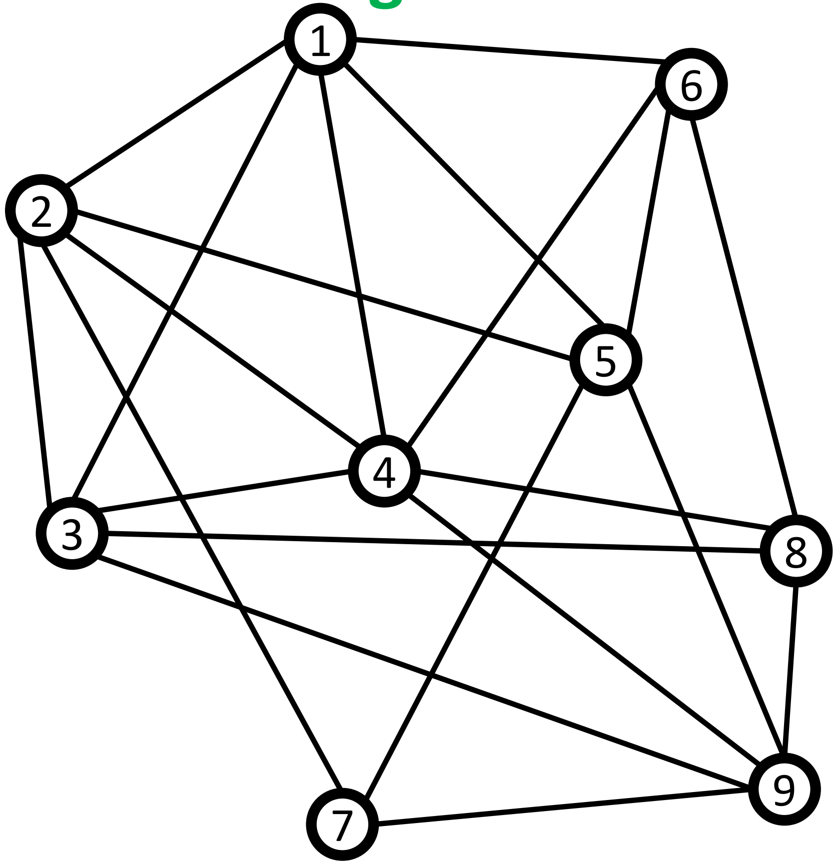
Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.



Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

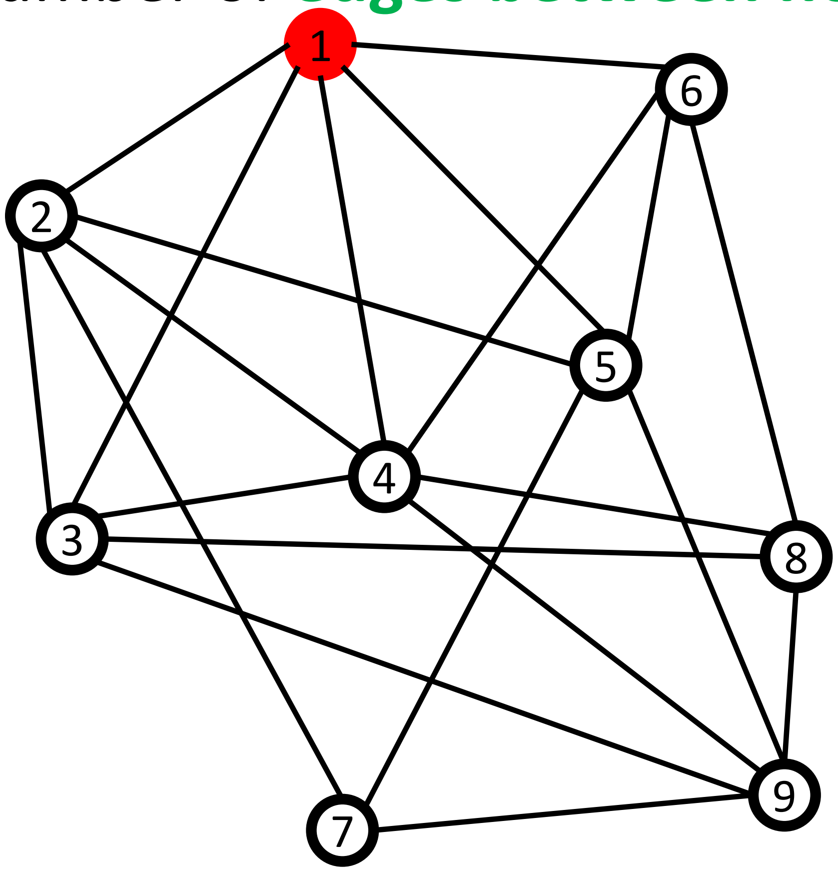


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

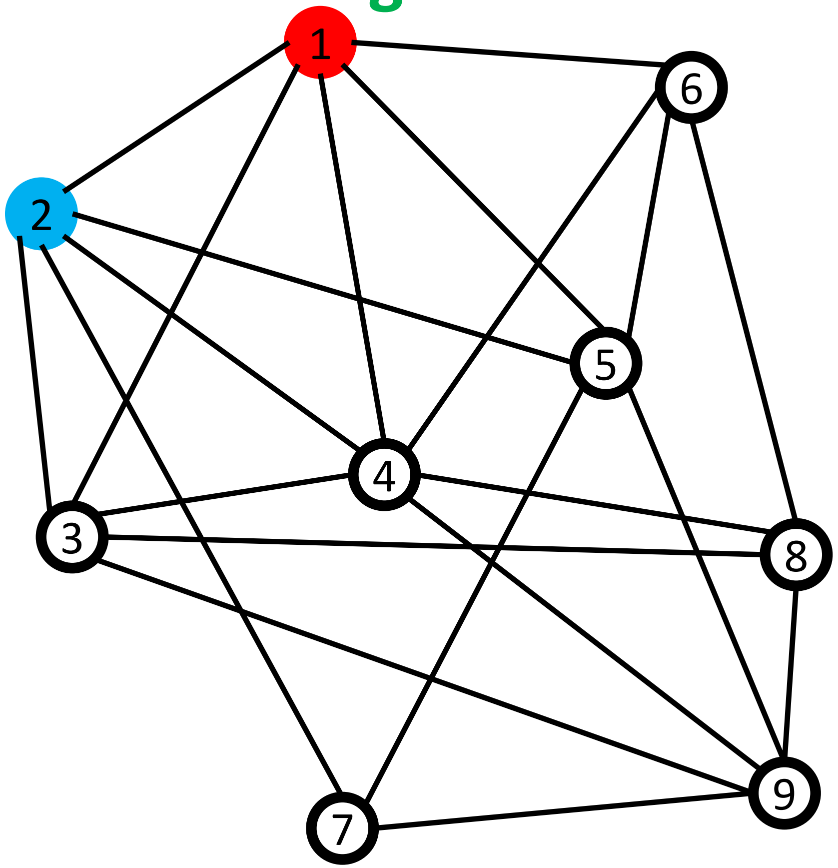


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

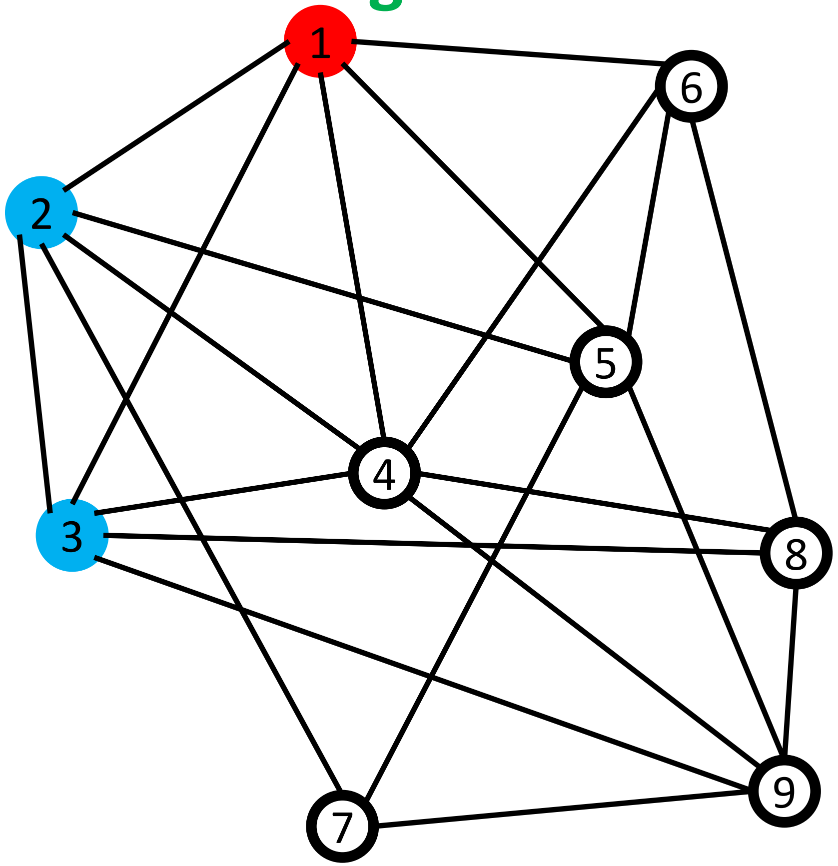


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

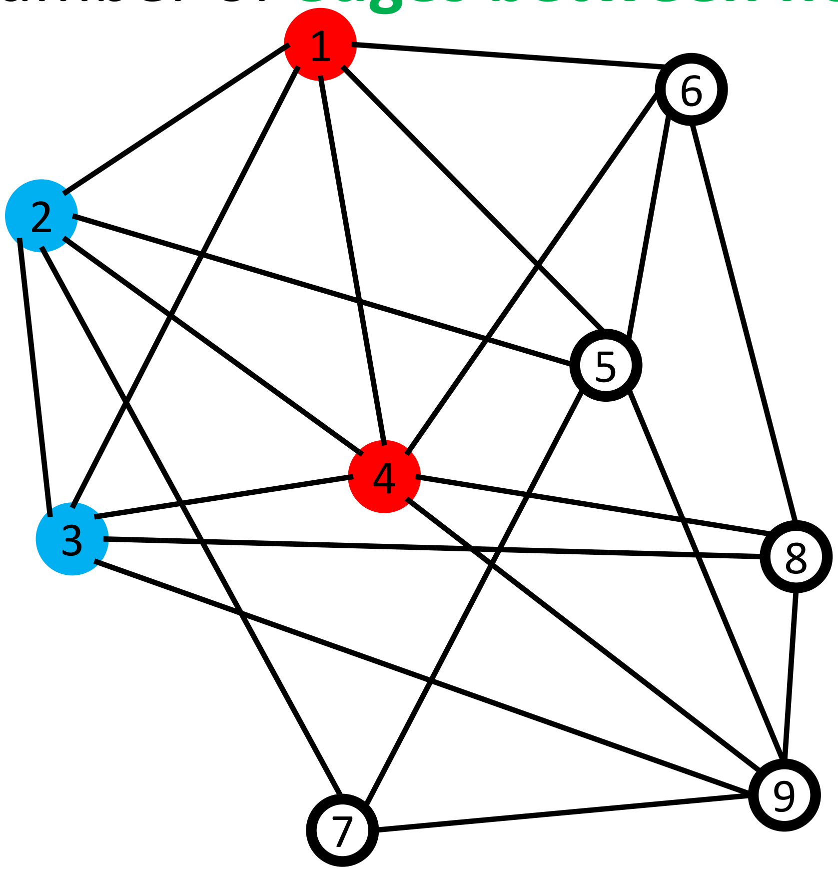


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

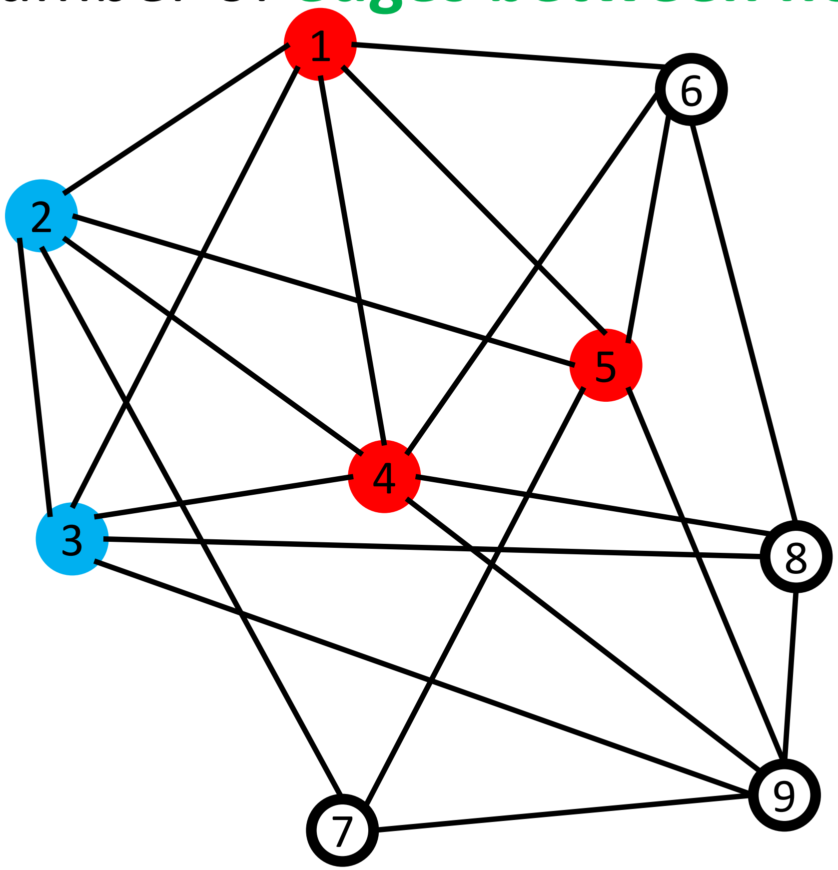


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

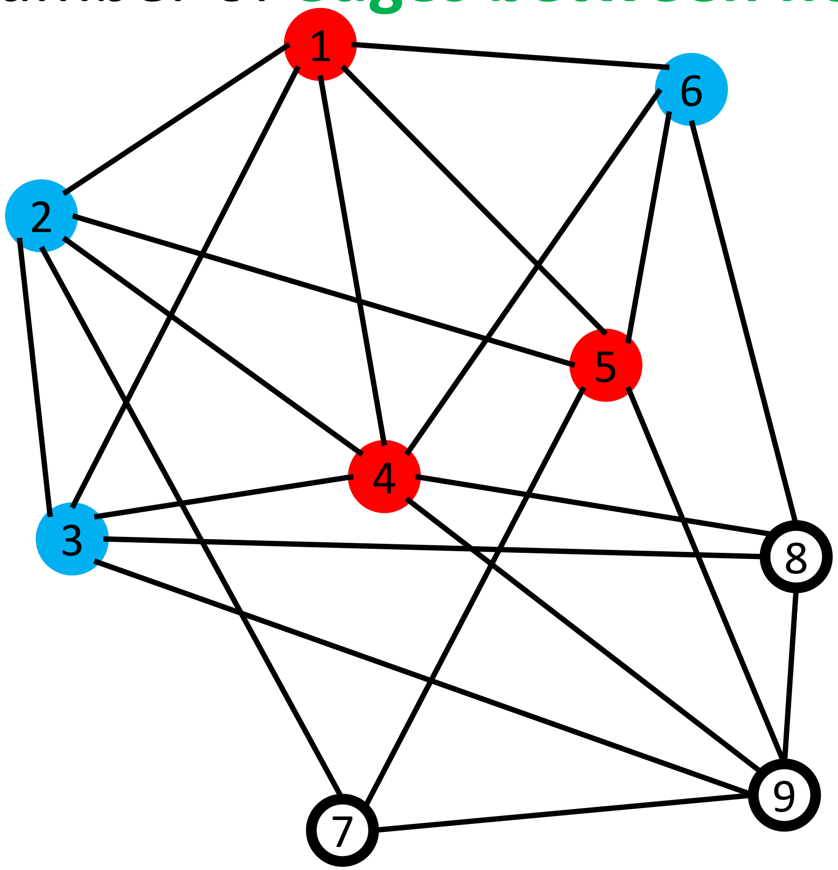


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

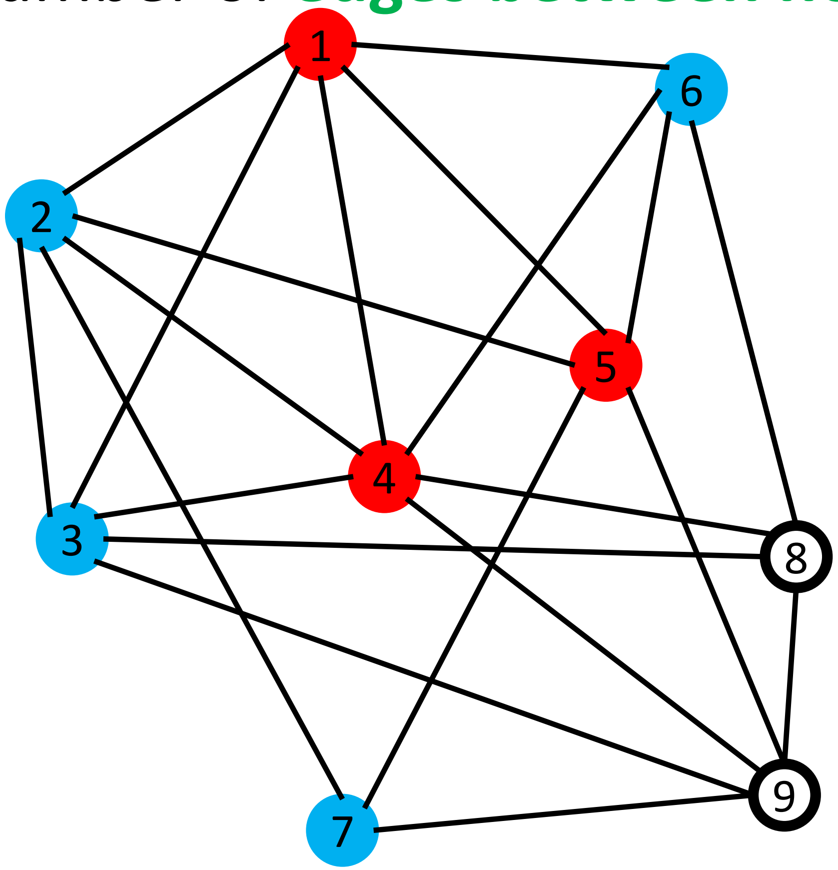


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

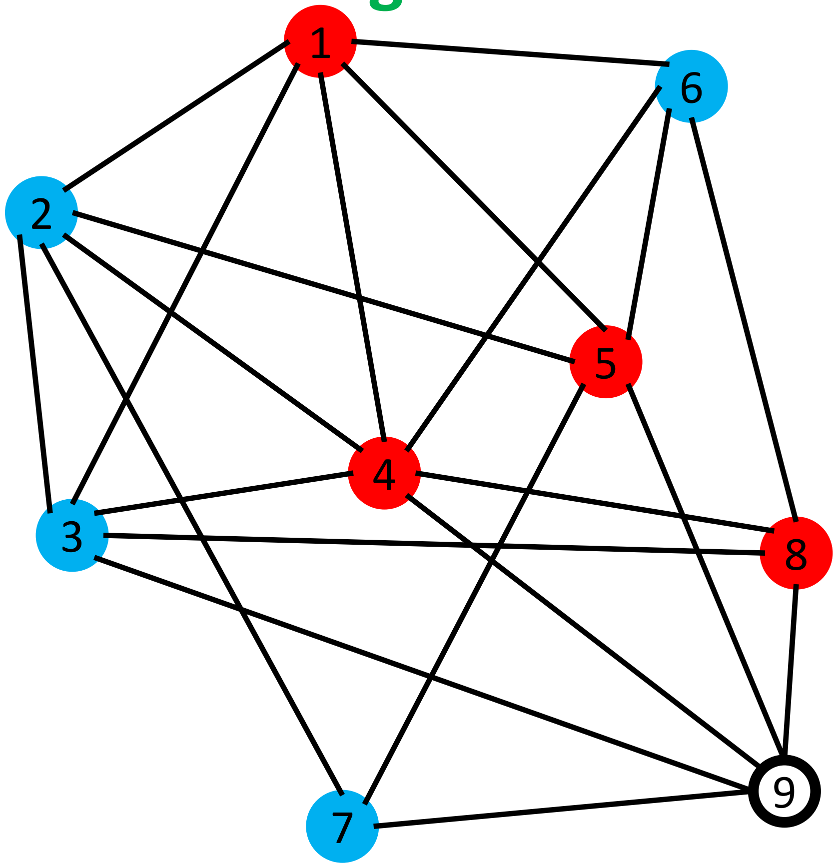


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

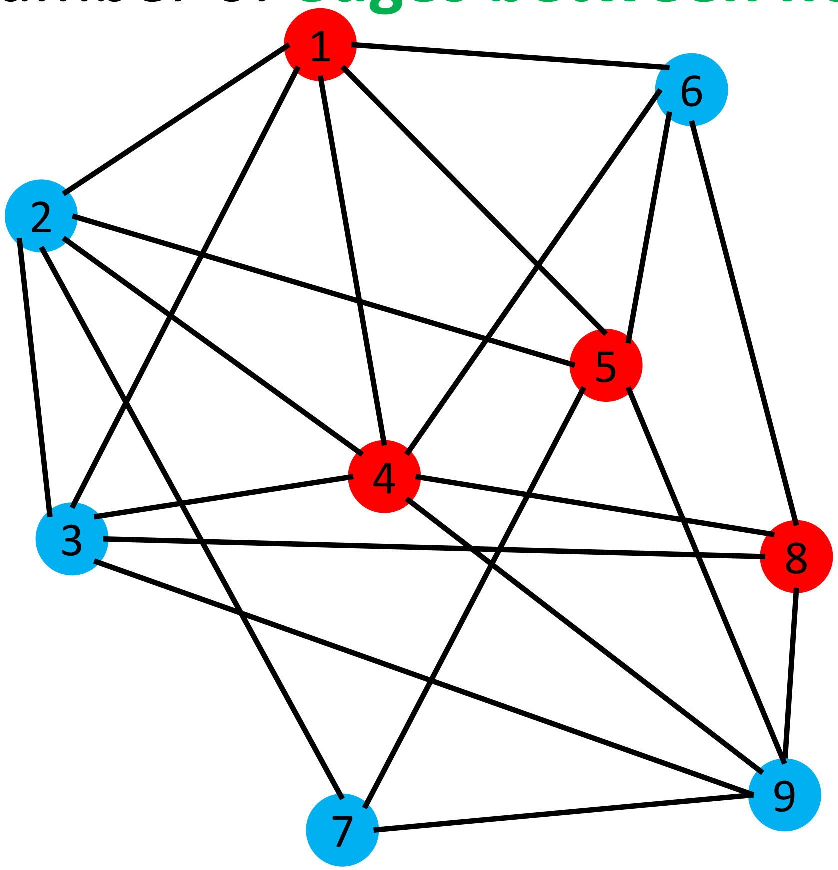


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

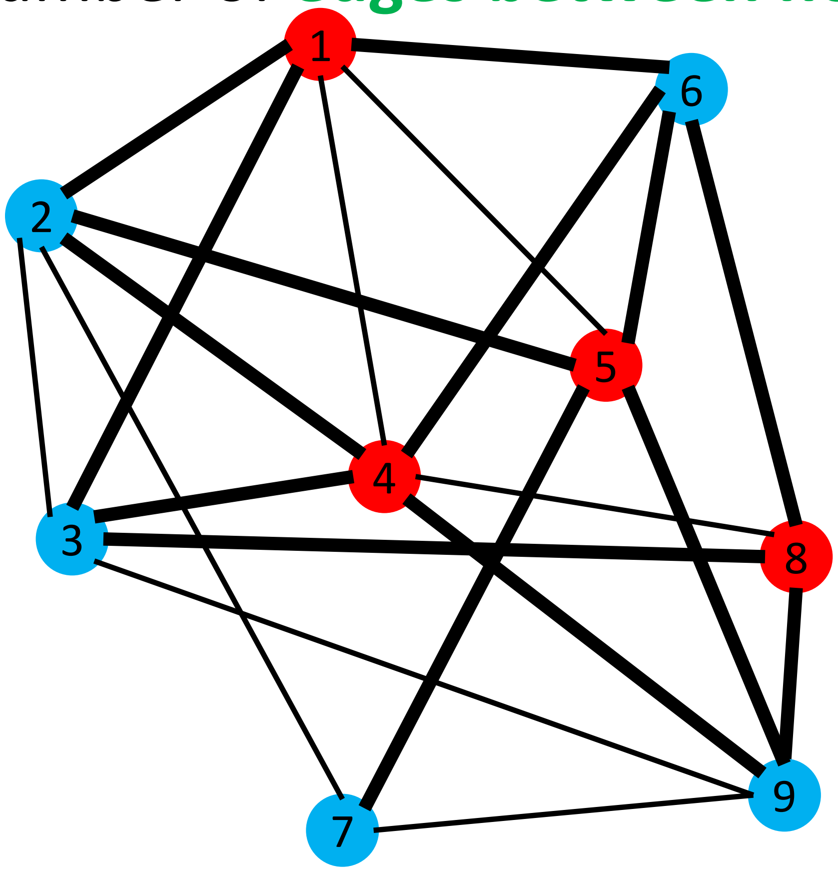


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

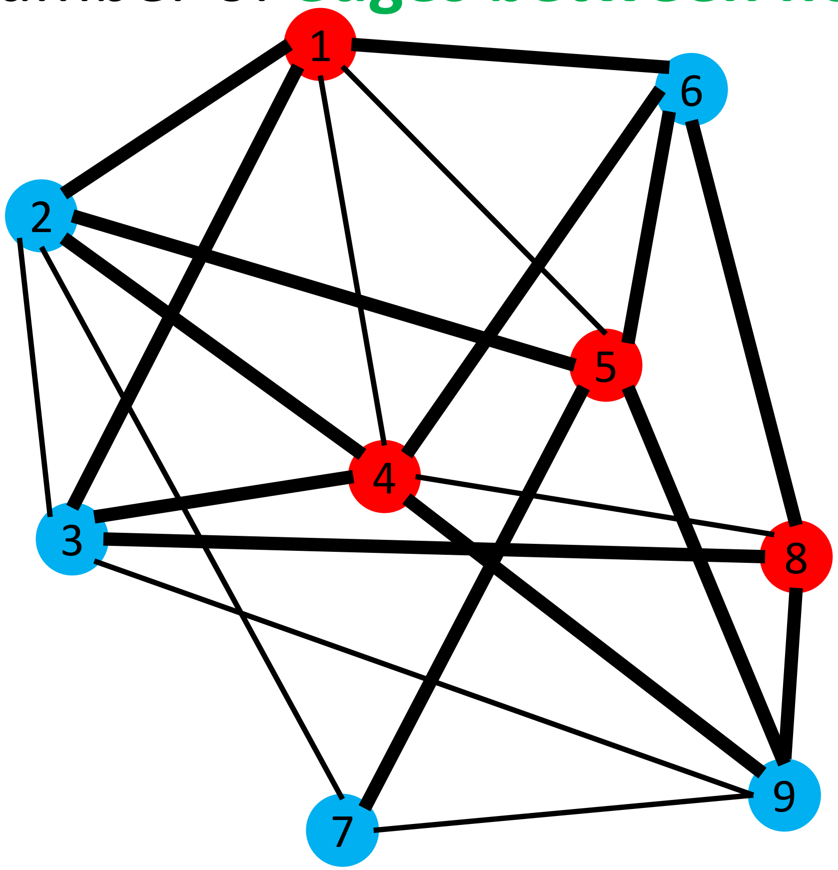


An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.



An algorithm:

- Examine the nodes in an arbitrary order
- For each node, decide its side in the bipartition **greedily**

The usual questions:

- **How good is this algorithm?**
- **Can we do better?**

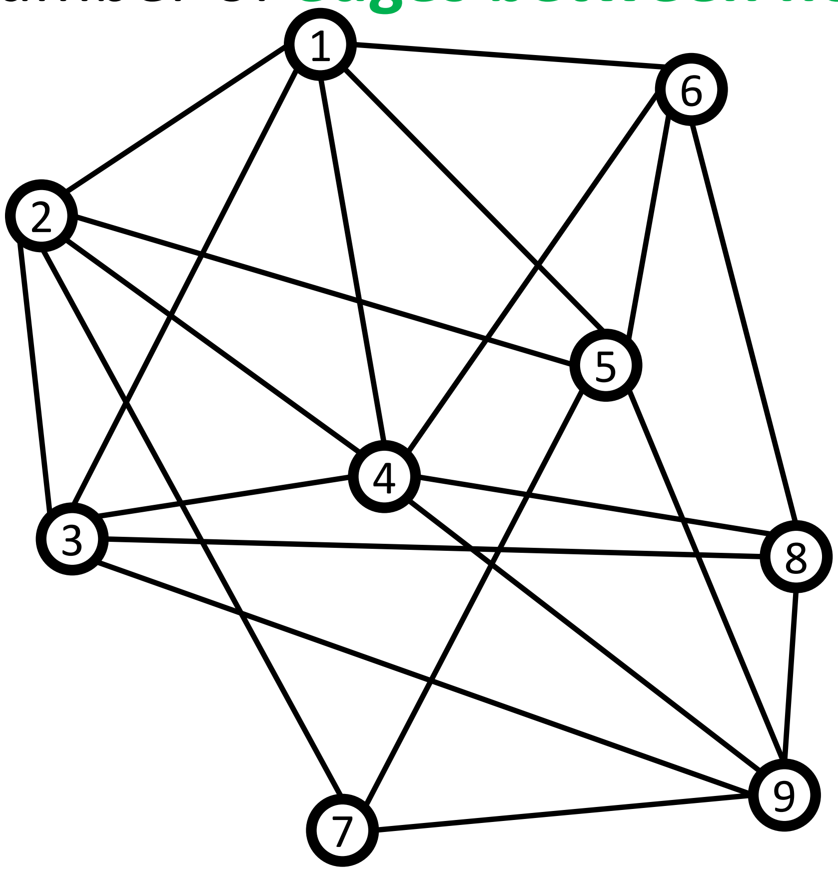
Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

- The problem is **NP-hard**, so we shouldn’t expect to find the largest cut quickly
- There are better algorithms (that provably compute large cuts) that are considerably more complicated

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

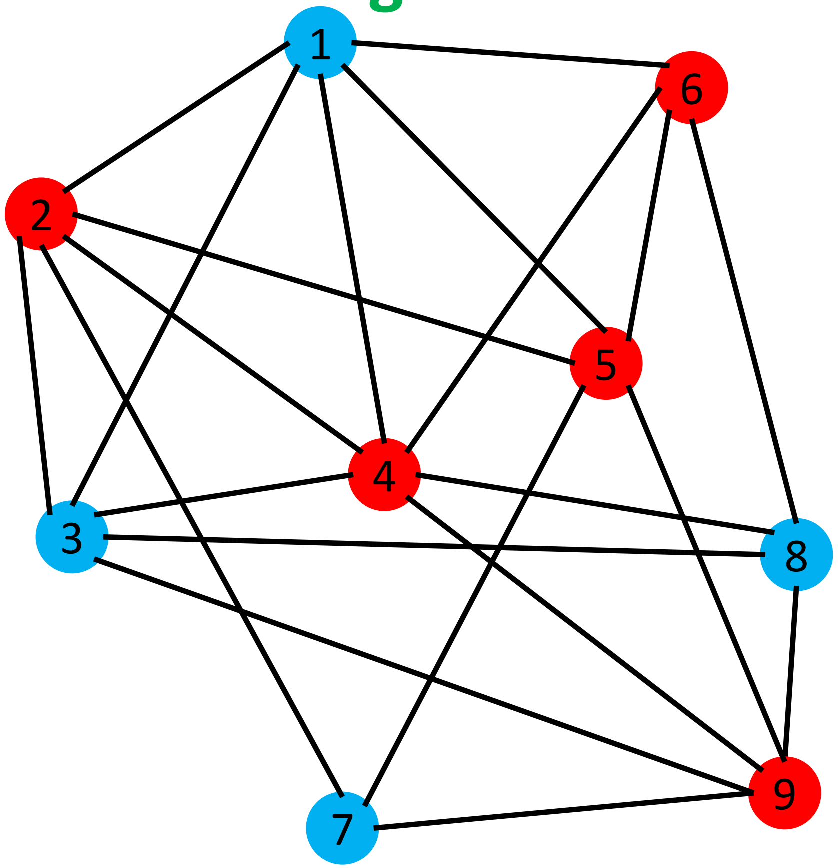


A much simpler algorithm:

- For each node, decide its side in the bipartition **randomly**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

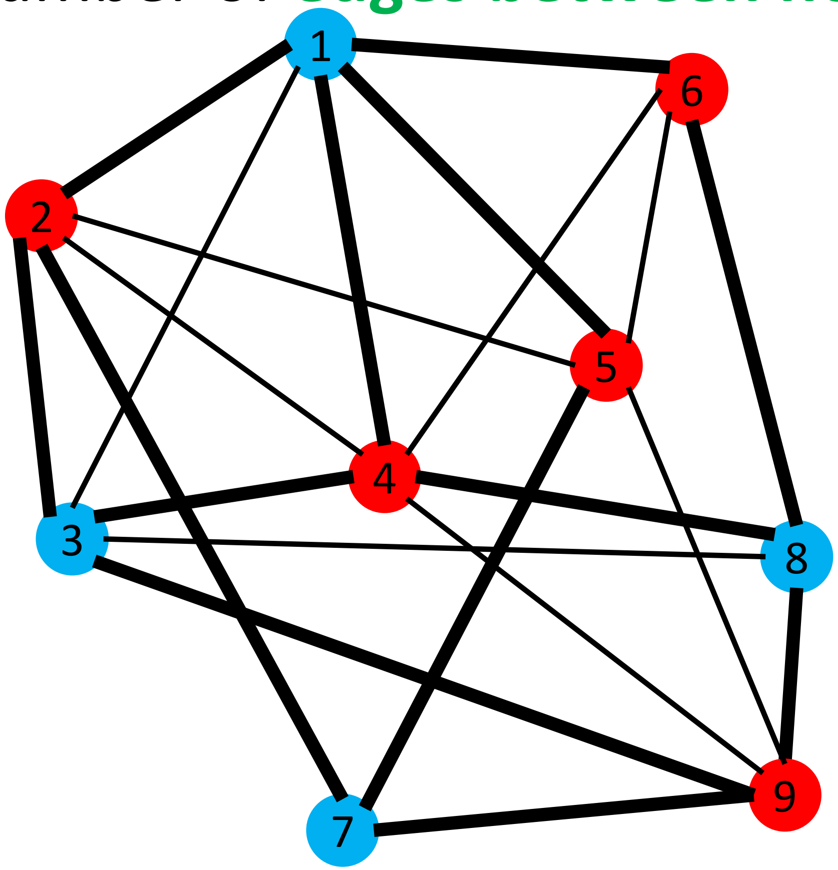


A much simpler algorithm:

- For each node, decide its side in the bipartition **randomly**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.

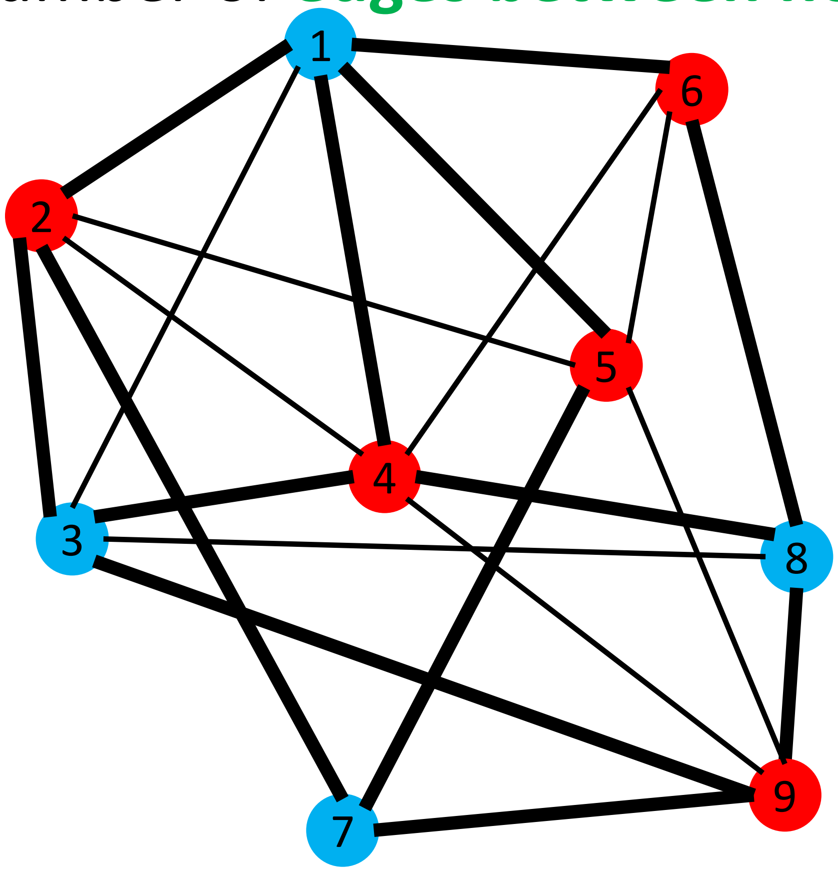


A much simpler algorithm:

- For each node, decide its side in the bipartition **randomly**

Example: Computing a large cut in a graph

- Given a graph, **partition the node set into two disjoint subsets** so that the total number of **edges between nodes at different subsets is maximized**.



A much simpler algorithm:

- For each node, decide its side in the bipartition **randomly**

With **very easy analysis**

- Each edge is in the cut with prob 1/2
- On average, half of the edges will be in the cut

Analysis (formally)

- For each edge $e \in E$, denote by X_e the random variable denoting **whether e is part of the cut**

$$X_e = \begin{cases} 0 & \text{if both endpoints of e are of the same color} \\ 1 & \text{otherwise} \end{cases}$$

- There are four possibilities on the colors of the endpoints of e (blue-blue, blue-red, red-blue, red-red)
- Each of them happens with probability 1/4
- Hence, $\mathbf{Pr}[\boldsymbol{X_e} = \mathbf{1}] = \mathbf{1/2}$

Analysis (formally)

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

Analysis (formally)

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

by definition

Analysis (formally)

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

by definition

by linearity of expectation

Analysis (formally)

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

by definition

by linearity of expectation

what is the expectation
of a 0/1 r.v.?

Analysis (formally)

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

by definition

by linearity of expectation

what is the expectation
of a 0/1 r.v.?

by previous slide

Analysis (formally)

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

by definition

by linearity of expectation

what is the expectation of a 0/1 r.v.?

by previous slide

The size of the max-cut cannot exceed the total number of edges

Example: Computing a large cut in a graph

- I.e., $\mathbb{E}[C] \geq OPT/2$
- So, the algorithm returns a cut that is (at least) **half-optimal on average**
- A better guarantee: return a cut that is **half-optimal with high probability**
- Possible: **repeat** the algorithm several times and keeping the largest cut
- The approximation guarantee of 1/2 is the **second best result we know** and is achieved by many algorithms (greedy, local-search, randomized, etc)
- Hard to imagine a **simpler** algorithm than our randomized one
- Best known algorithm has approximation guarantee 0.878 and uses **semidefinite programming**

Example: in a graph

linearity of expectation

- I.e., $\mathbb{E}[C] \geq OPT/2$
- So, the algorithm returns a cut that is (at least) **half-optimal on average**
- A better guarantee: return a cut that is **half-optimal with high probability**
- Possible: **repeat** the algorithm several times and keeping the largest cut
- The approximation guarantee of 1/2 is the **second best result we know** and is achieved by many algorithms (greedy, local-search, randomized, etc)
- Hard to imagine a **simpler** algorithm than our randomized one
- Best known algorithm has approximation guarantee 0.878 and uses **semidefinite programming**

Example:

linearity of expectation

concentration bounds

- I.e., $\mathbb{E}[C] \geq OPT/2$
- So, the algorithm returns a cut that is (at least) **half-optimal on average**
- A better guarantee: return a cut that is **half-optimal with high probability**
- Possible: **repeat** the algorithm several times and keeping the largest cut
- The approximation guarantee of 1/2 is the **second best result we know** and is achieved by many algorithms (greedy, local-search, randomized, etc)
- Hard to imagine a **simpler** algorithm than our randomized one
- Best known algorithm has approximation guarantee 0.878 and uses **semidefinite programming**

Example:

linearity of expectation

concentration bounds

Markov inequality

- I.e., $\mathbb{E}[C] \geq OPT/2$
- So, the algorithm retu ... -o timal on average
- A better guarantee ... eturn a cut that is **half-optimal with high probability**
- Possible: **repeat** the algorithm several times and keeping the largest cut
- The approximation guarantee of 1/2 is the **second best result we know** and is achieved by many algorithms (greedy, local-search, randomized, etc)
- Hard to imagine a **simpler** algorithm than our randomized one
- Best known algorithm has approximation guarantee 0.878 and uses **semidefinite programming**

Quicksort

Problem: Sorting

- Input: An array of n numbers, in arbitrary order
- Output: An array of the same numbers, sorted from smallest to largest

- Example (input)

3	8	2	5	1	4	7	6

- Output:

1	2	3	4	5	6	7	8

Quicksort: Main idea

- Recursive calls to a fast subroutine for partial sorting
- Step 1: select a pivot element
- Step 2: reorganize around the pivot

3	8	2	5	1	4	7	6

Quicksort: Main idea

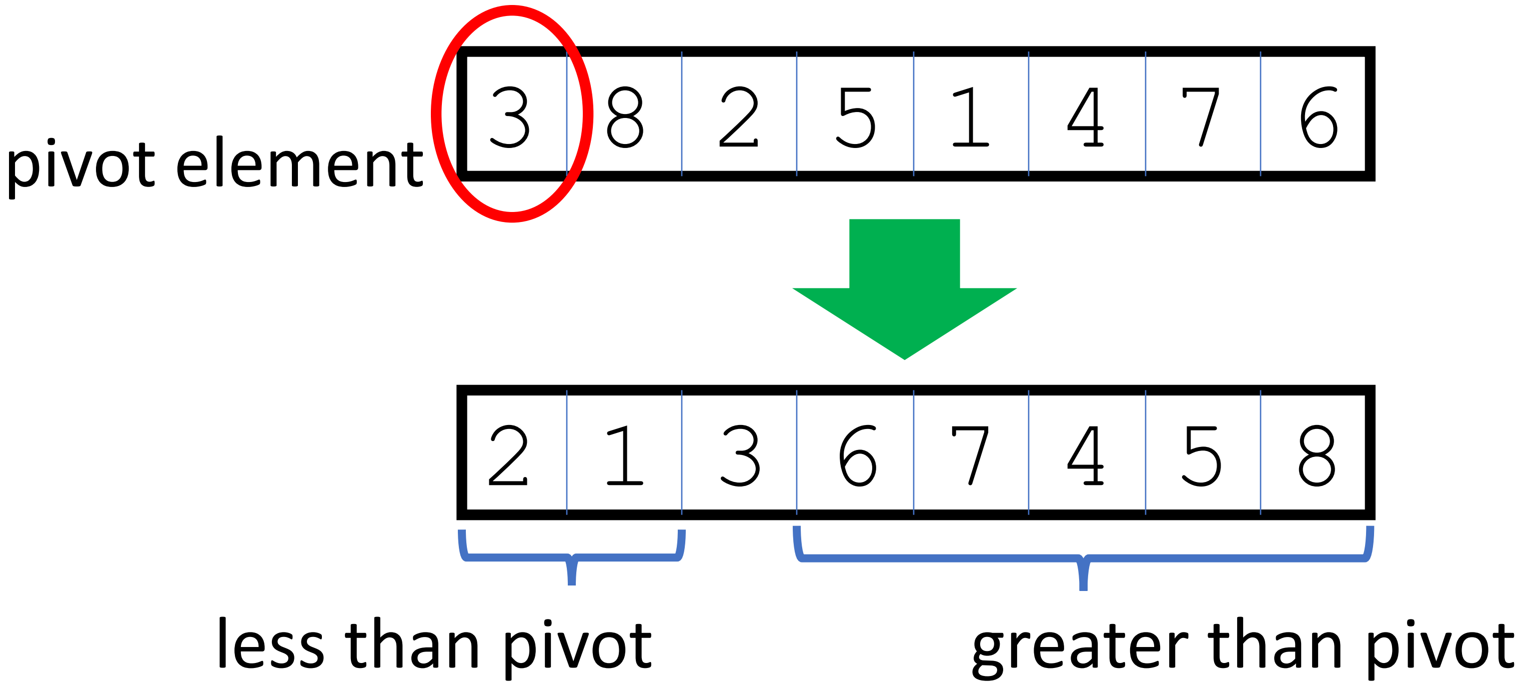
- Recursive calls to a fast subroutine for partial sorting
- Step 1: select a pivot element
- Step 2: reorganize around the pivot

pivot element

3	8	2	5	1	4	7	6

Quicksort: Main idea

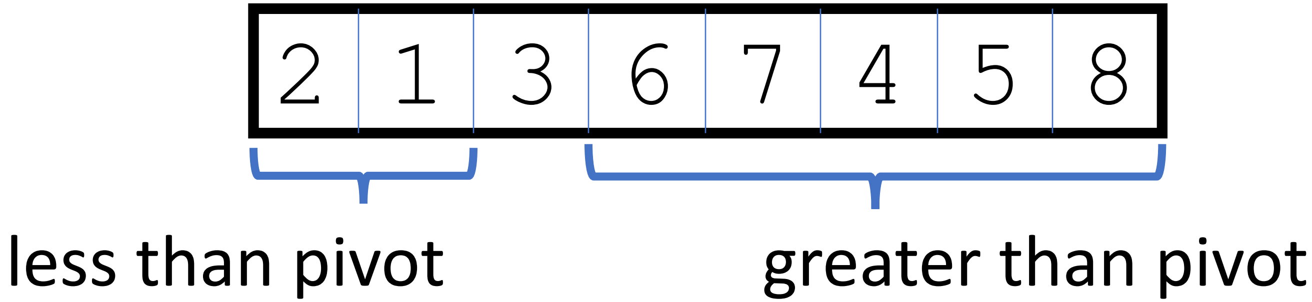
- Recursive calls to a fast subroutine for partial sorting
- Step 1: select a pivot element
- Step 2: reorganize around the pivot



Quicksort: Main idea

- Recursive calls to a fast subroutine for partial sorting
- Step 1: select a pivot element
- Step 2: reorganize around the pivot

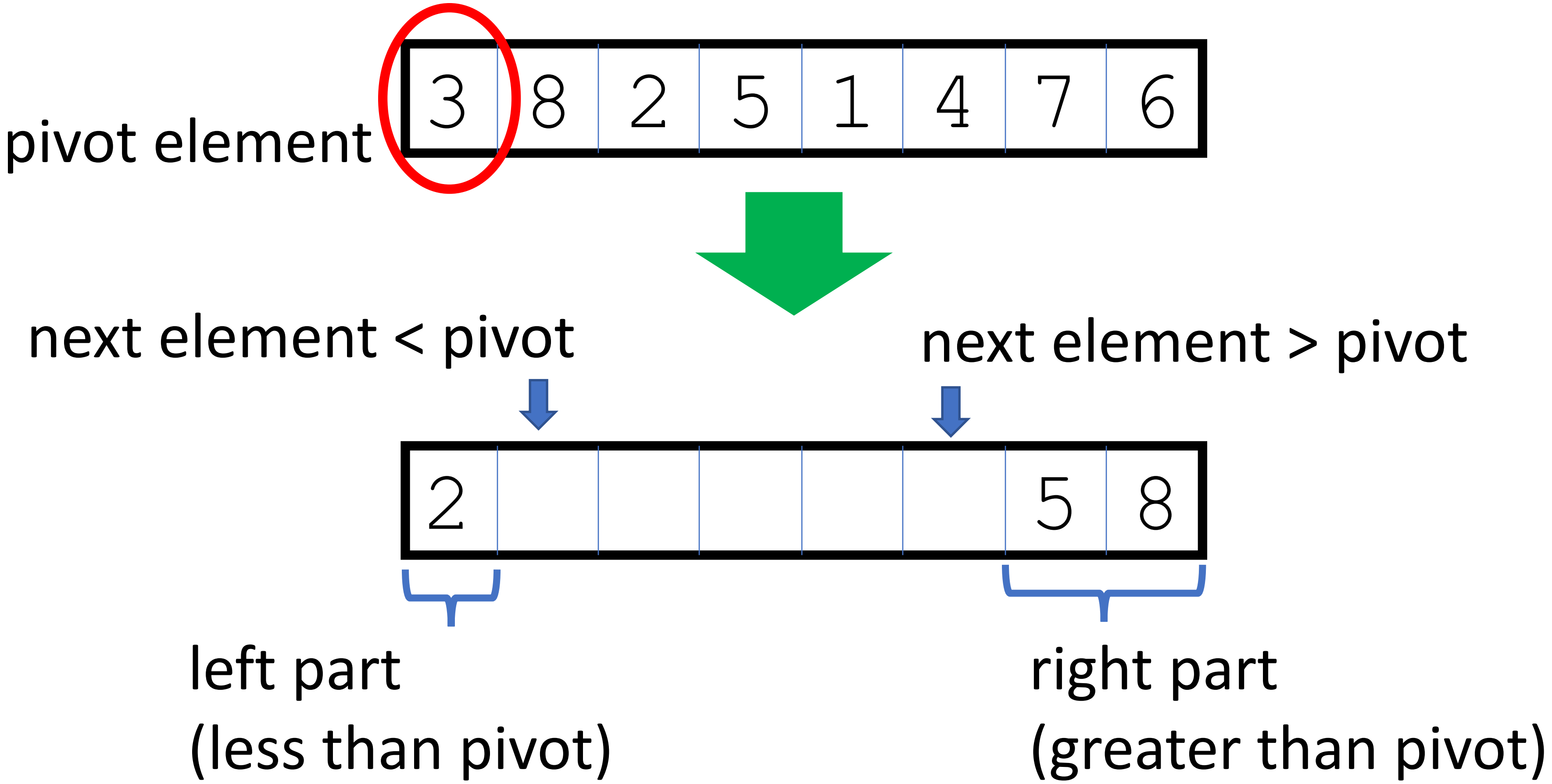
- The subroutine takes O(n) steps and makes significant progress
- The pivot element is in the final position
- The sorting problem is reduced to two smaller sorting problems



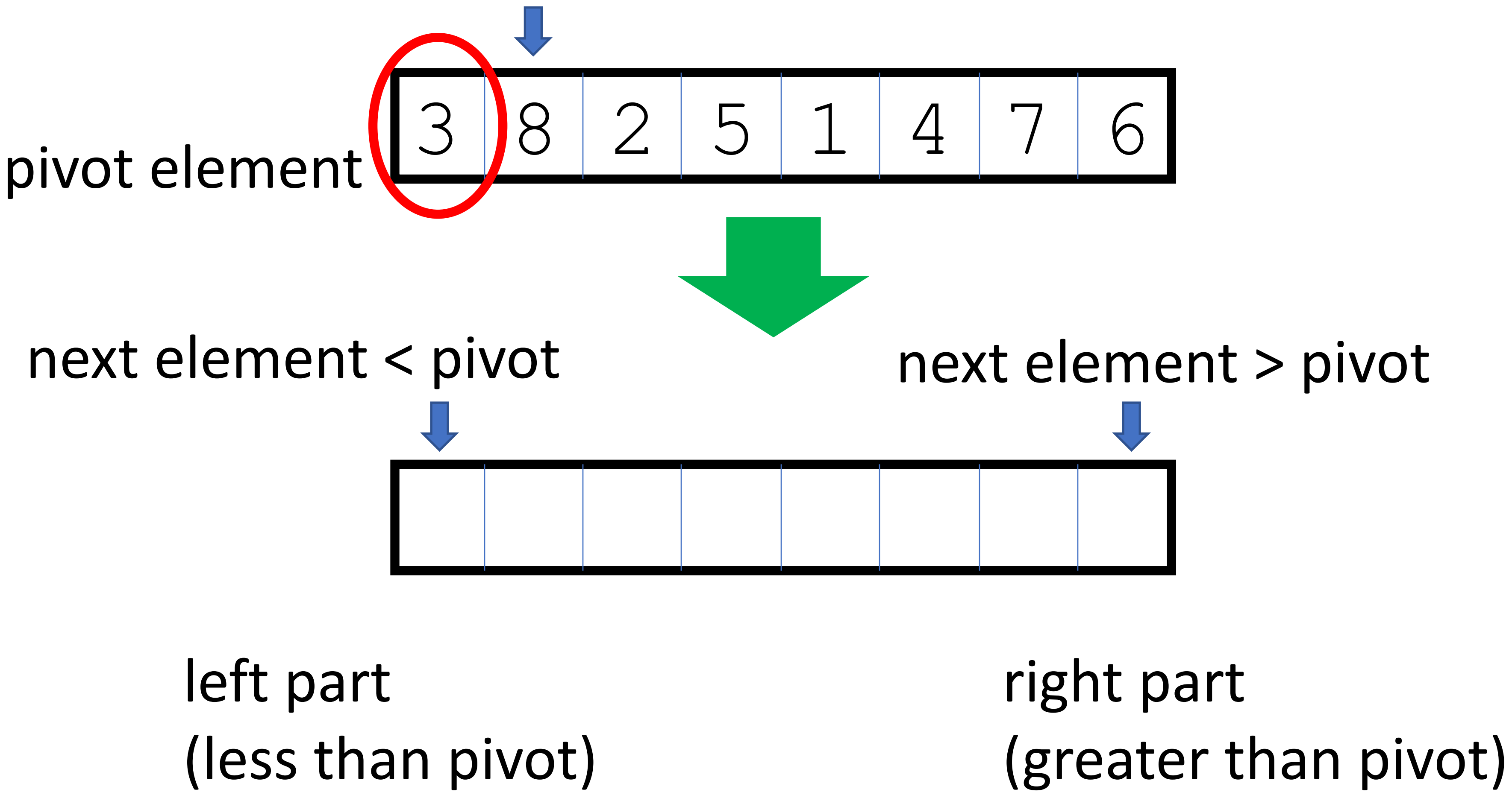
Quicksort: a high-level description

- If $n \leq 1$ then return // already sorted
- Choose a **pivot** element p // to be implemented
- Partition A around p // easy to implement
- Recursively sort the **left** part of A
- Recursively sort the **right** part of A

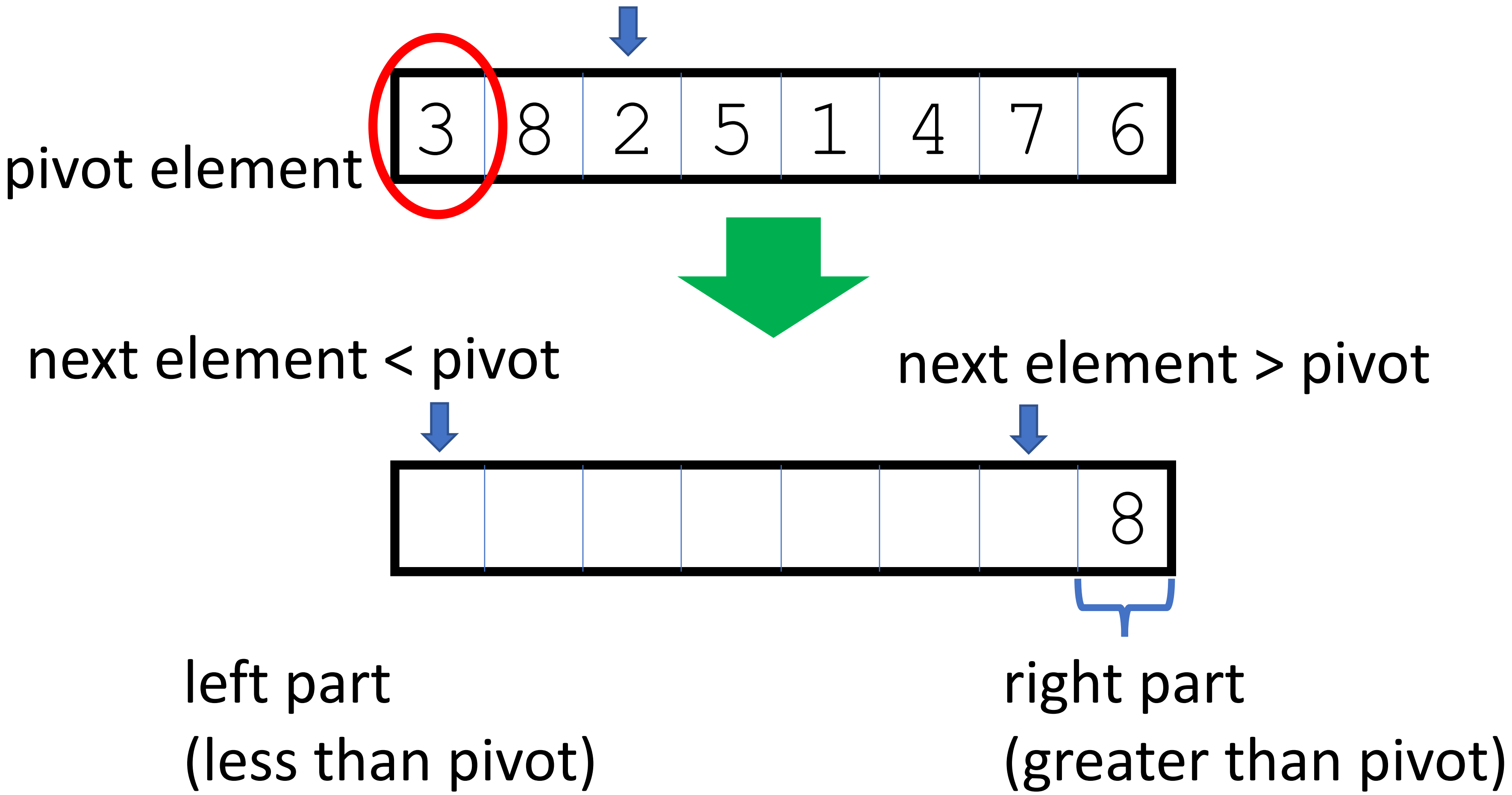
Partition A around p: an easy implementation



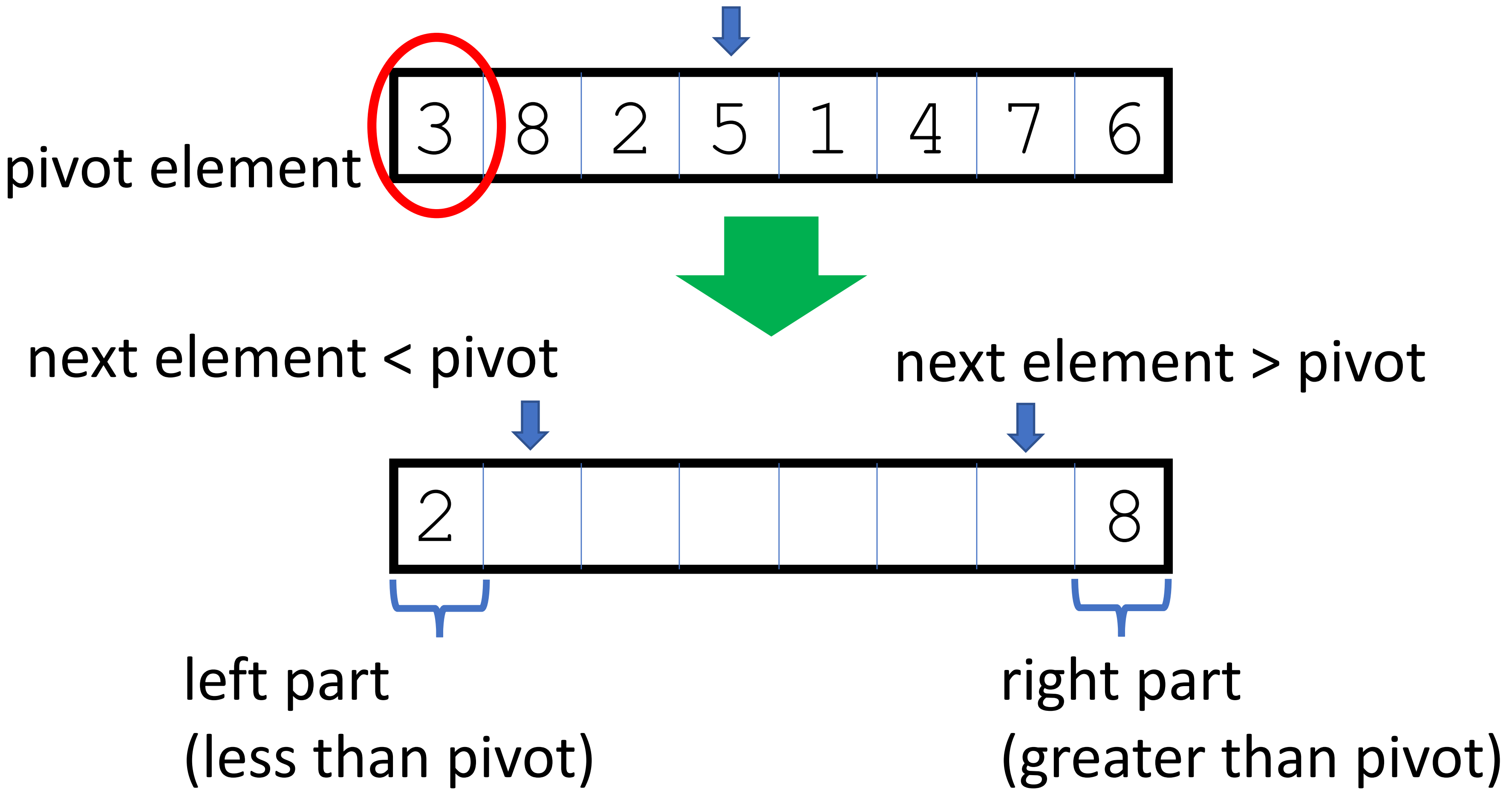
Partition A around p: an easy implementation



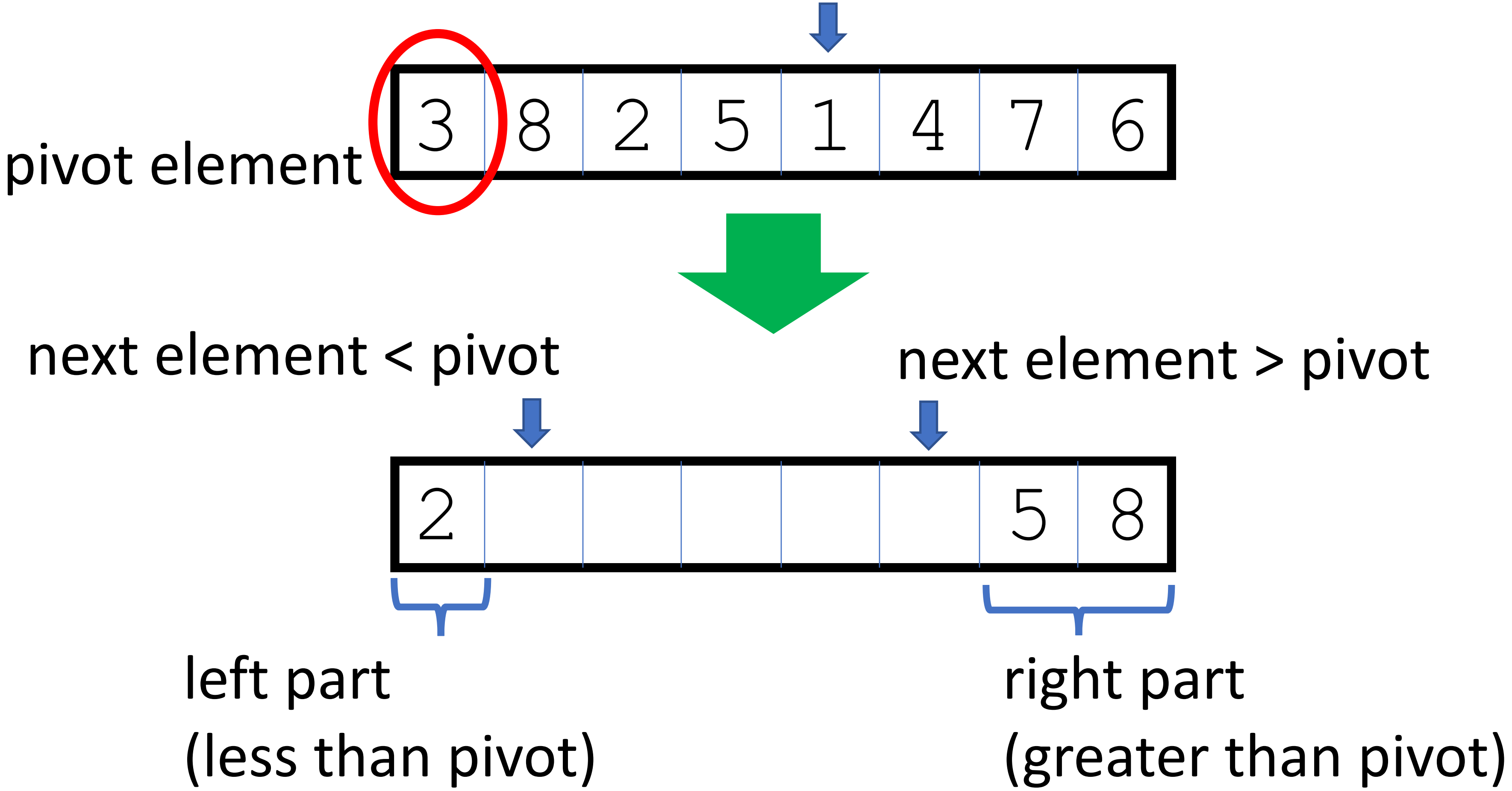
Partition A around p: an easy implementation



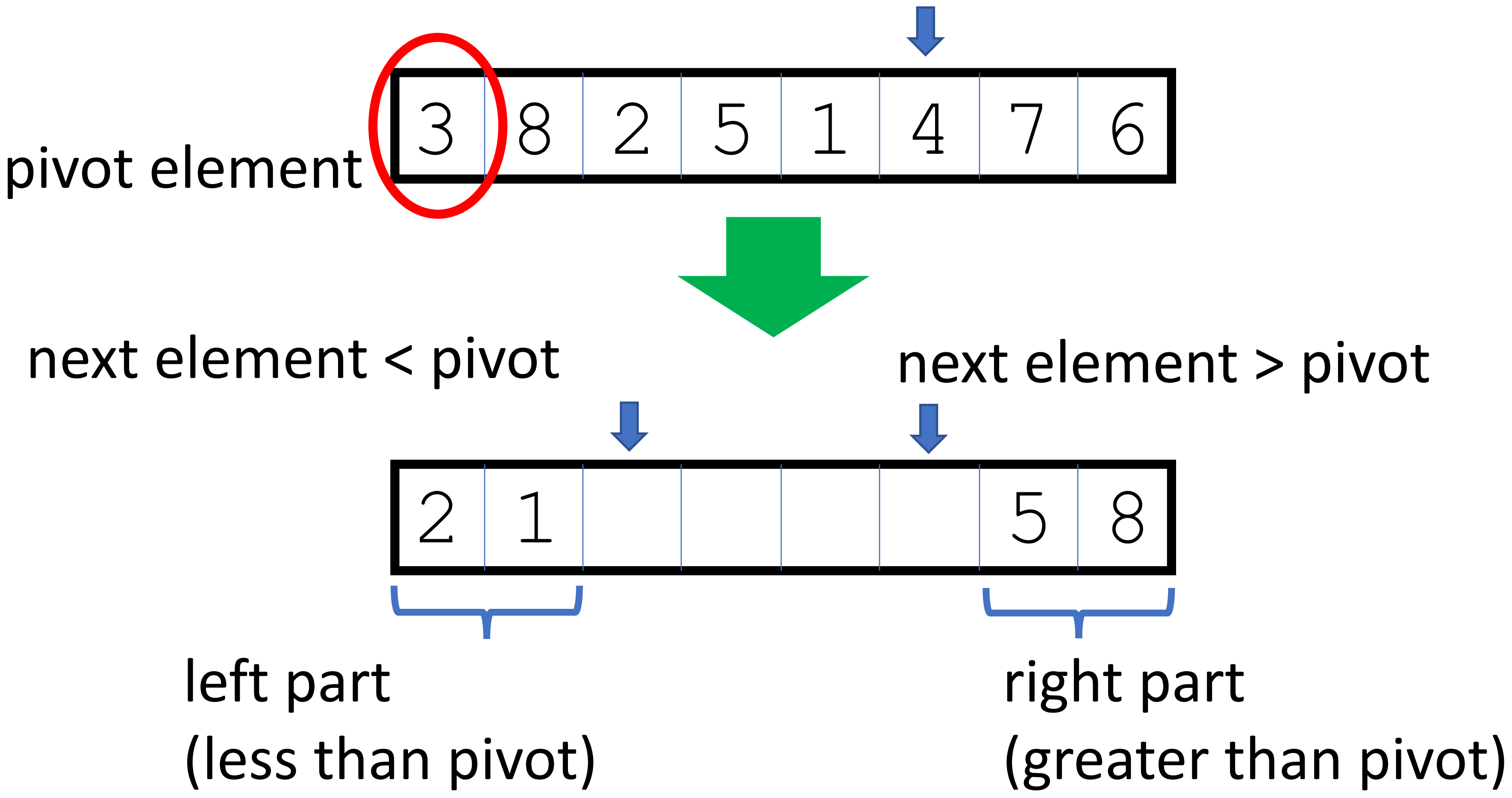
Partition A around p: an easy implementation



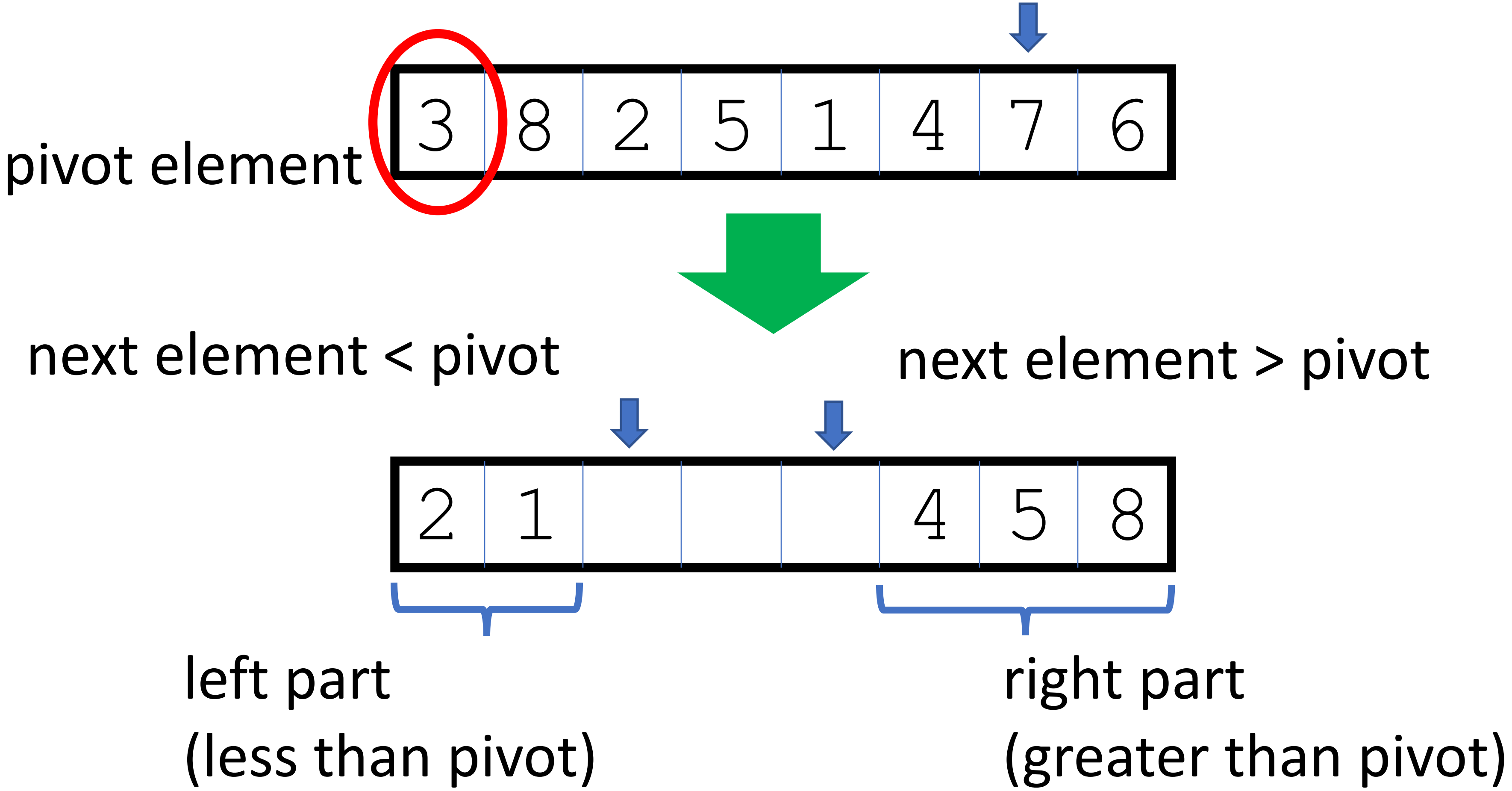
Partition A around p: an easy implementation



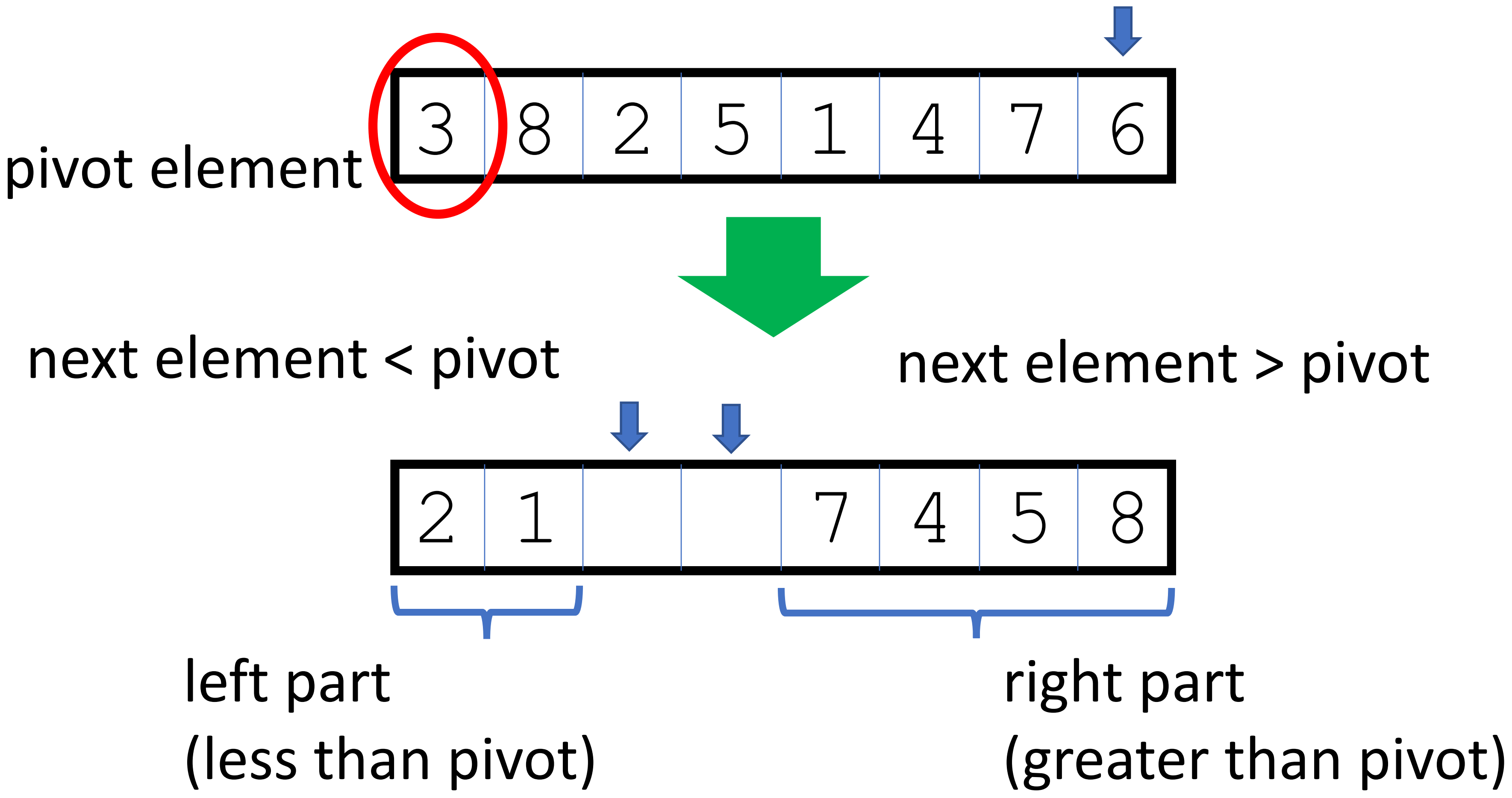
Partition A around p: an easy implementation



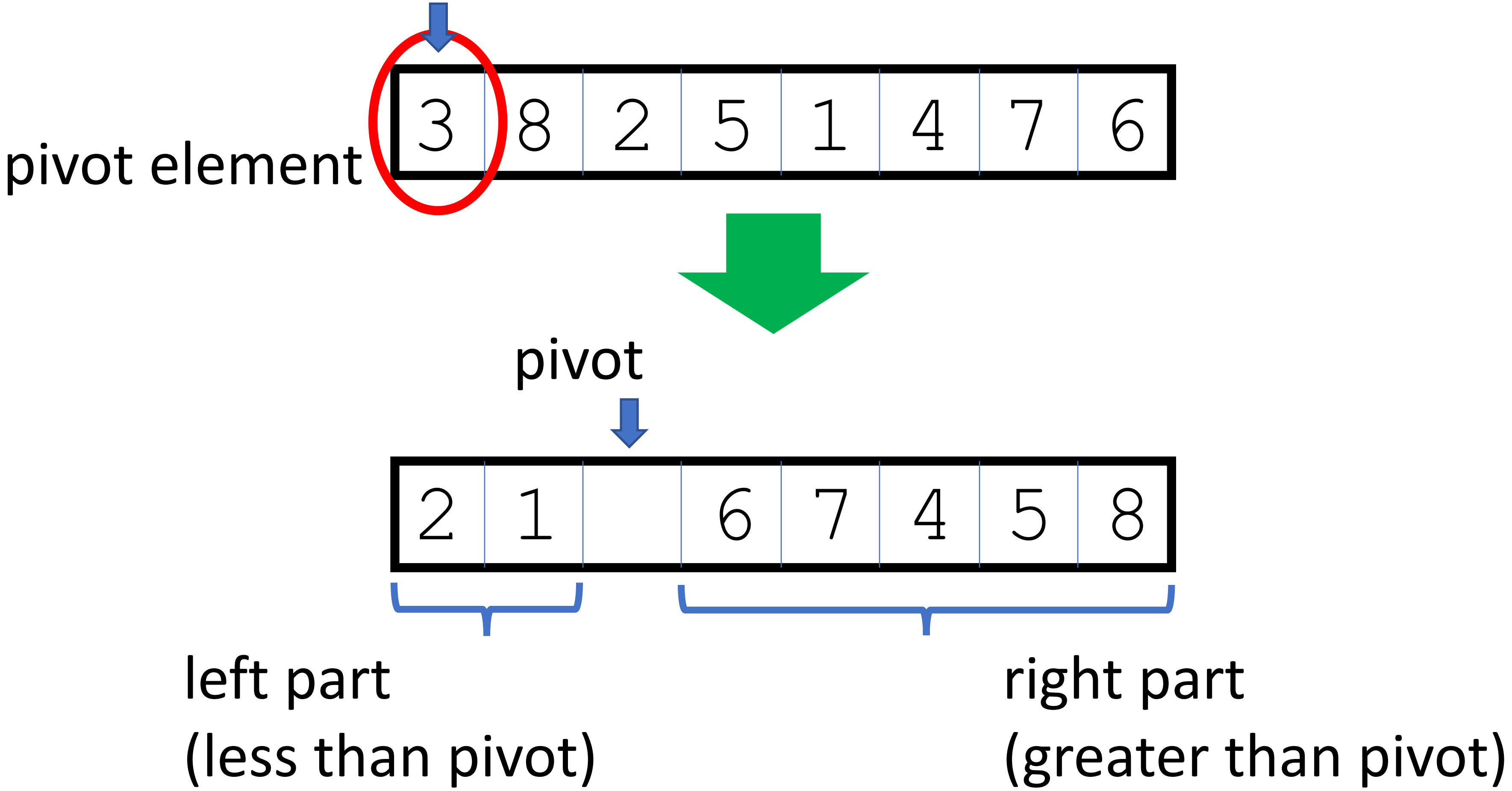
Partition A around p: an easy implementation



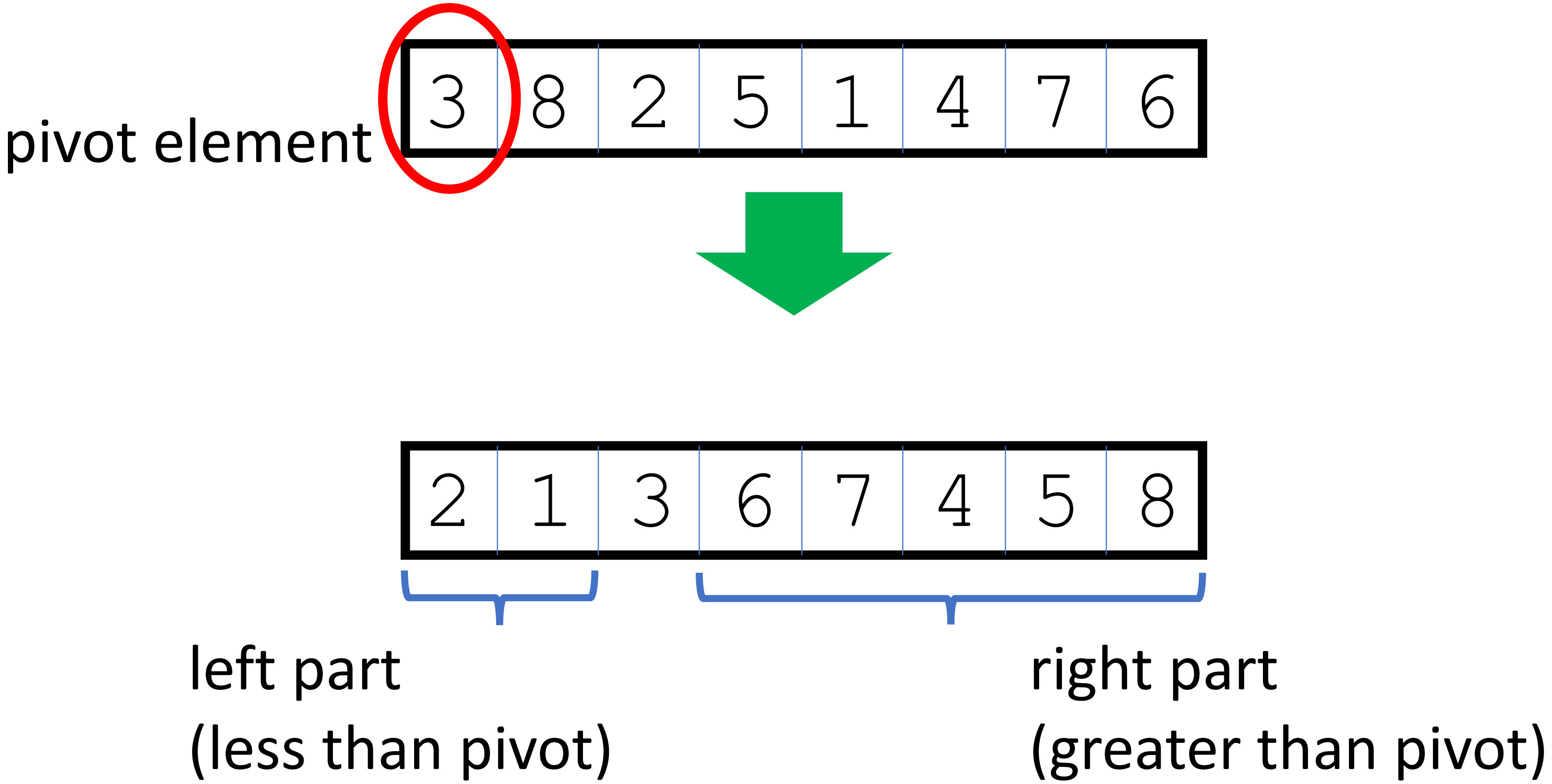
Partition A around p: an easy implementation



Partition A around p: an easy implementation



Partition A around p: an easy implementation



Pseudocode for Quicksort (first try)

- Input: array A of n distinct elements, left and right endpoint $l, r \in \{1, 2, \dots, n\}$
- Output: elements of subarray $A[l], A[l+1], \dots, A[r]$ are sorted from smallest to largest

if $l \geq r$ **then return**

$j :=$ **partition** (A, l, r) // reorganize subarray using $A[l]$ as pivot

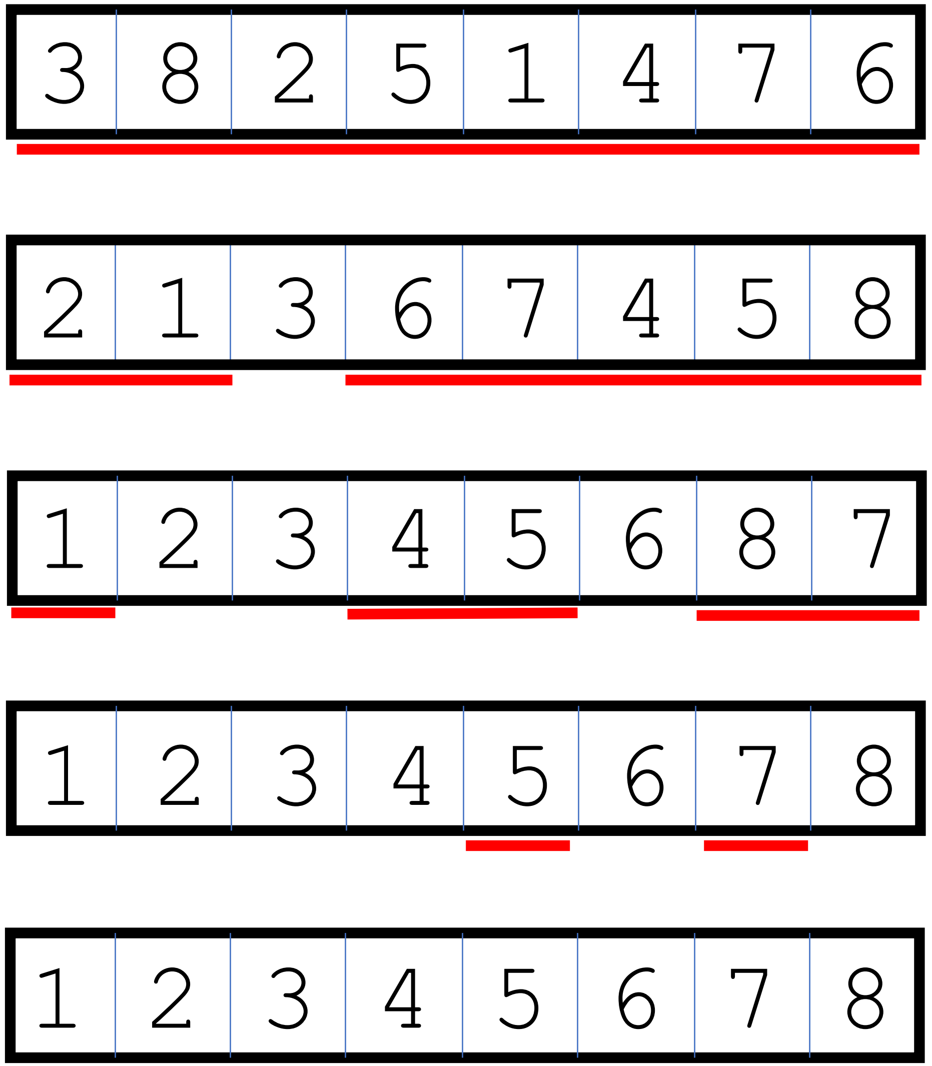
quicksort $(A, l, j-1)$

quicksort $(A, j+1, r)$

- Quicksort is revoked by calling **quicksort** $(A, 1, n)$

How good is our implementation so far?

input



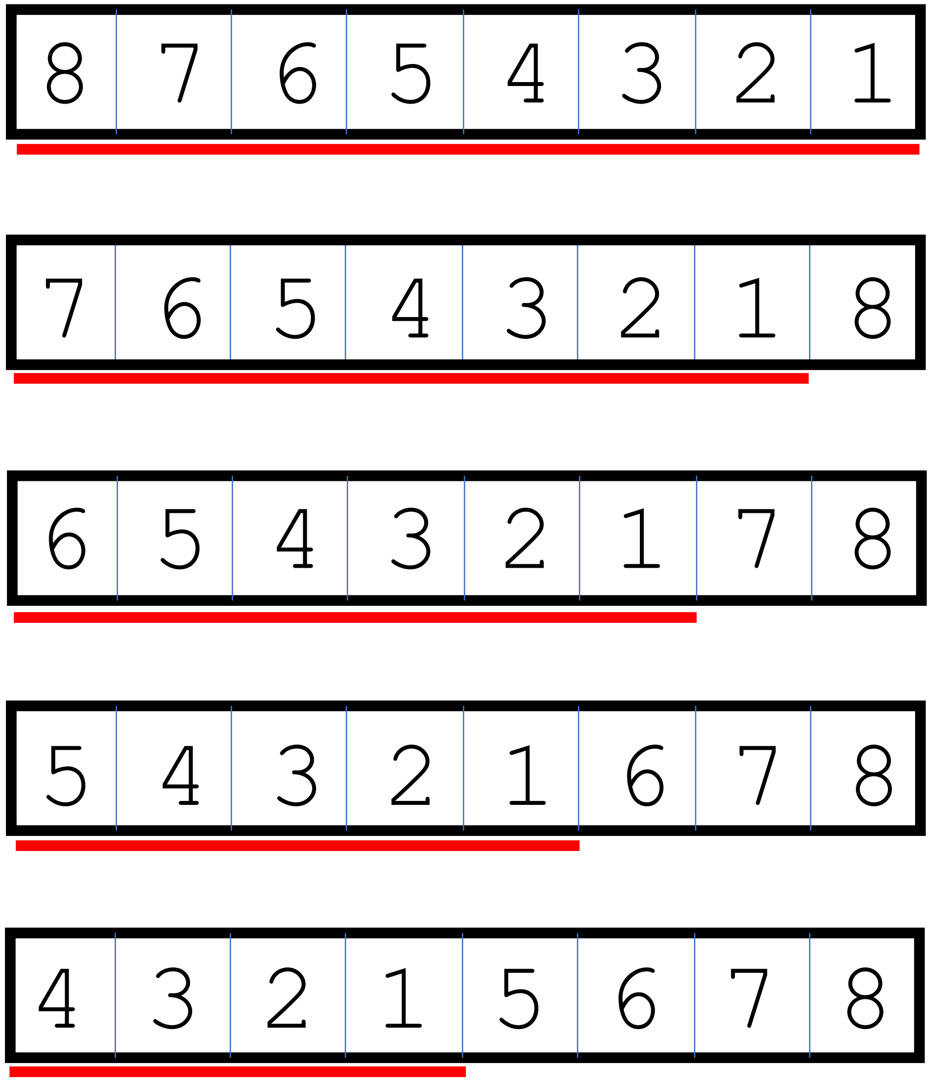
Red intervals indicate the total amount of work (running time)

Our current implementation does not look bad on this particular instance

Is it always so?

How good is our implementation so far?

input



Red intervals indicate the total amount of work (running time)

Clearly $\Theta(n^2)$ time

Improving our implementation

- Spending $O(n)$ for reorganizing the subarray around the pivot seems necessary
- Improvements seem possible by selecting a **better pivot**

Improving our implementation

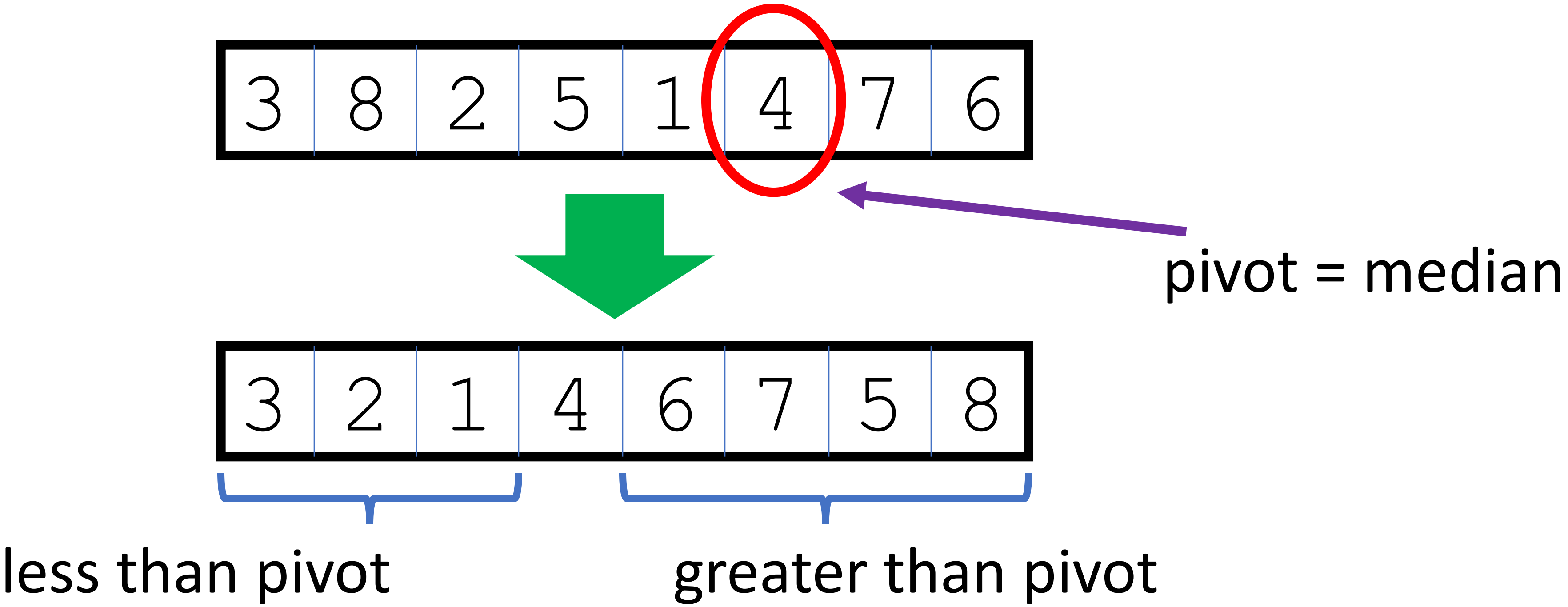
- Input: array A of n distinct elements, left and right endpoint $l, r \in \{1, 2, \ldots, n\}$
- Output: elements of subarray $A[l], A[l+1], \ldots, A[r]$ are sorted from smallest to largest

if $l \geq r$ **then return**
i := **pivot** (A, l, r) // select the pivot element **cleverly**
Swap $A[l]$ with $A[i]$ // put the pivot in the **first position** of the subarray
j := **partition** (A, l, r) // reorganize subarray using $A[l]$ as pivot
quicksort $(A, l, j-1)$
quicksort $(A, j+1, r)$

- Quicksort is revoked by calling **quicksort** $(A, 1, n)$

Killer idea

- Select the median of $A[l], A[l+1], \ldots, A[r]$ as pivot



How good is our implementation so far?

input

8	7	6	5	4	3	2	1

3	2	1	4	5	6	7	8

1	2	3	4	5	6	8	7

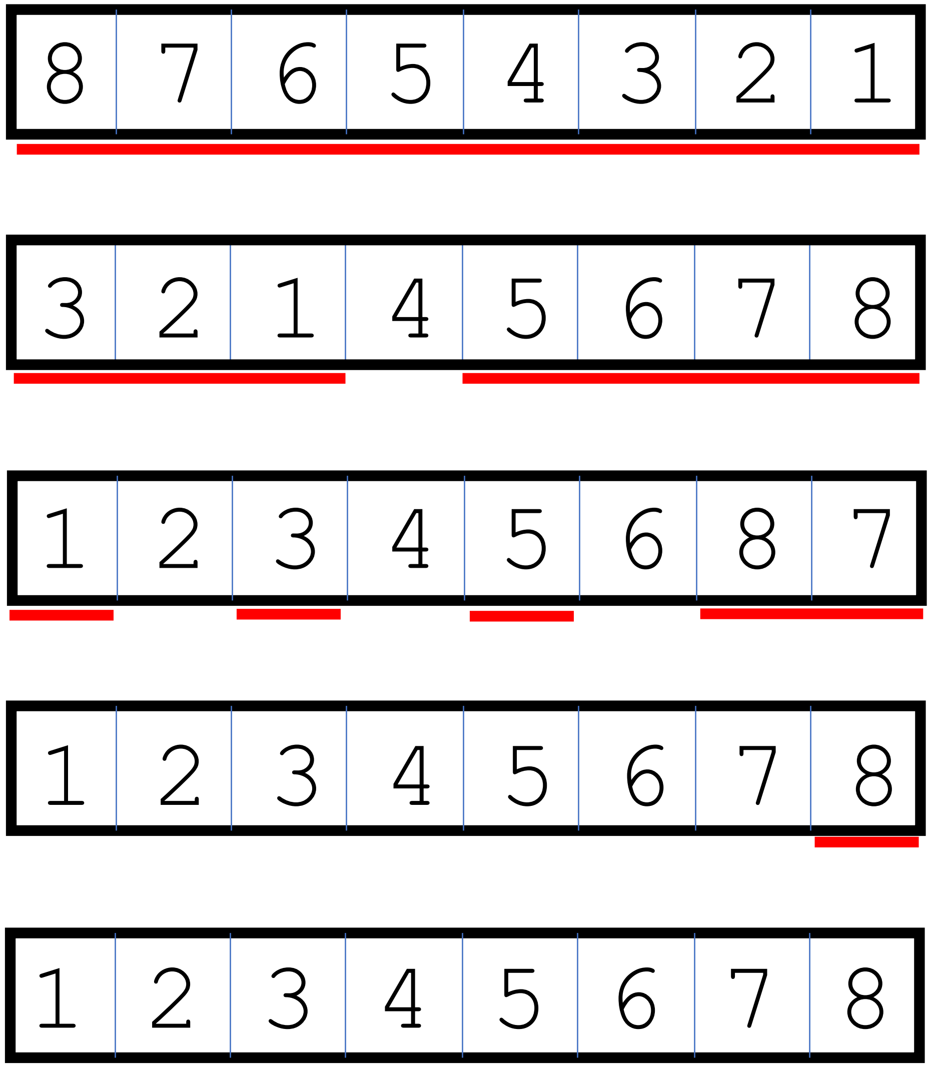
1	2	3	4	5	6	7	8

1	2	3	4	5	6	7	8

Red intervals indicate the total amount of work (running time)

How good is our implementation so far?

input



Red intervals indicate the total amount of work (running time)

So far, we ignore the additional time to find the median

How good is our implementation so far?

- Denote by $T(n)$ the running time of the algorithm on input of size n

$$T(n) = 2 \cdot T(n/2) + T_{\text{partition}}(n) + T_{median}(n)$$

- Clearly, $T_{\text{partition}}(n) = O(n)$
- There exists a linear-time (deterministic) algorithm for finding the median
- Using this algorithm to implement **`pivot`**(), we get

$$T(n) = 2 \cdot T(n/2) + O(n)$$

which yields $T(n) = O(n \log n)$

How good is our implementation so far?

- Denote by $T(n)$ the runn

$$T(n) = 2 \cdot$$

Personal opinion: this is the most beautiful among the basic algorithms!

- Clearly, $T_{\text{partition}}(n) = O(n)$
- There exists a linear-time (deterministic) algorithm for finding the median
- Using this algorithm to implement **`pivot`**(), we get

$$T(n) = 2 \cdot T(n/2) + O(n)$$

which yields $T(n) = O(n \log n)$

How good is Quicksort as described so far?

Advantages:

- Best possible running time (like Mergesort)
- **In-place** implementation (low space, unlike Mergesort)

Disadvantages:

- Complicated (Mergesort is simpler)

Randomized Quicksort

A randomized implementation

- Input: array A of n distinct elements, left and right endpoint $l, r \in \{1, 2, \dots, n\}$
- Output: elements of subarray $A[l], A[l+1], \dots, A[r]$ are sorted from smallest to largest

if $l \geq r$ **then return**

i := **r_pivot** (A, l, r) // choose the pivot from $\{l, \dots, r\}$, **uniformly at random**

Swap $A[l]$ with $A[i]$

j := **partition** (A, l, r)

quicksort $(A, l, j-1)$

quicksort $(A, j+1, r)$

- Quicksort is revoked by calling **quicksort** $(A, 1, n)$

Why are random pivots good?

- The probability that the maximum element is always selected as the pivot is **negligible**
- The probability that the median is always selected as the pivot is also not large, but …
- On average, a random pivot gives us a **25%-75%** split
- Would be great if we could conclude to a

$$T(n) = T(3n/4) + T(n/4) + O(n)$$

  recursive relation for the running time, but this **does not make much sense** formally

Why are random pivots good?

- The probability that the maximum element is always selected as the pivot is **negligible**
- The probability that the median i... t... large, but …
- On average, a random pivot give...
- Would be great if we could conclude

$$T(n) = T(3n/4) + T(n/4) + O(n)$$

recursive relation for the running time, but this **does not make much sense** formally

This yields $T(n) = O(n \log n)$
as well

Analysis of Randomized Quicksort

Analysis: preliminaries

Fix an input array with n elements

We will just bound the **expected number** of **comparisons in the calls of `partition`()**

Why?

- This involves all operations besides the selection of pivots
- Typically, the uniform selection of a pivot among n elements can be implemented **with $O(\log n)$ coin tosses, on average**
- So, the time for selecting pivots is at most $O(n \log n)$

Notation

- Given the input array, denote by z_i the ***i*-th smallest** element
- For every pair of indices in $\{1, 2, \dots, n\}$ with $i < j$, define X_{ij} as the r.v. denoting the number of times the elements z_i and z_j get compared by Quicksort
- X_{ij} is 1 if both z_i and z_j belong to the same subarray when some of them is selected as pivot, otherwise it is 0
- Then, the number of comparisons in the calls of **`partition`**() is

$$C = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Notation

This is to be expected☺
No good sorting algorithm should
perform the same comparison twice

- Given the in
- For every pair of i ... $_{ij}$ as the r.v. denoting the nu... times the elements z_i and z_j get compared by Quicksort
- X_{ij} is 1 if both z_i and z_j belong to the same subarray when some of them is selected as pivot, otherwise it is 0
- Then, the number of comparisons in the calls of **`partition`**() is

$$C = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Notation

This is to be expected☺
No good sorting algorithm should perform the same comparison twice

- Given the in
- For every pair o $_{ij}$ as the r.v. denoti ts z_i and z_j get compared by Q
- X_{ij} subarray when some of them is selecte
- Then, the number of com ons in the calls of **`partition`**() is

Recall that we have to bound the expectation of C

$$C = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Bounding the expectation of C

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

Bounding the expectation of C

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

by our definitions

Bounding the expectation of C

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

by our definitions

by linearity of expectation

Bounding the expectation of C

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

by our definitions

by linearity of expectation

what is the expectation of a 0/1 r.v.?

Bounding the expectation of C

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

by our definitions

by linearity of expectation

what is the expe n
of a 0/1 r.v.?

So, we need to understand
$\Pr[X_{ij} = 1]$

Déjà vu: Analysis of our max-cut algorithm

- Denote by C the size of the **cut returned by the algorithm**
- C is a random variable with $C = \sum_{e \in E} X_e$

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = \sum_{e \in E} \Pr[X_e = 1] = |E|/2 \geq OPT/2$$

by definition

by linearity of expectation

what is the expectation of a 0/1 r.v.?

by previous slide

The size of the max-cut cannot exceed the total number of edges

Computing $\Pr[X_{ij} = 1]$

Warming up: $\Pr[X_{1n} = 1] = ?$

Three cases for the first call of Quicksort on the whole array

- z_1 is selected as the pivot. Happens with prob. 1/n. Then, in **`partition`**(), z_n is compared to the pivot z_1 to decide z_n's position at the left or at the right of the pivot
- z_n is selected as the pivot. Happens with prob. 1/n. Similarly, **`partition`**() compared z_1 to the pivot z_n
- Neither z_1 nor z_n is selected as pivot. Then, **`partition`**() will put z_1 and z_n in different subarrays and they will never be compared
- Hence, $\Pr[X_{1n} = 1] = 2/n$

Computing $\Pr[X_{ij} = 1]$

- For general $i < j$, observe that z_i and z_j get compared at some step of the execution of Quicksort **if and only if** one of them is chosen as a pivot before any of $z_{i+1}, z_{i+2}, \dots, z_{j-1}$
- Hence,

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$$

Why?

- As long as none of the elements $z_i, z_{i+1}, z_{i+2}, \dots, z_{j-1}, z_j$ is selected as pivot, they all belong to the same subarray
- At some point, some of them will be selected as pivot

Final push

Recall that

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

and $\Pr\left[X_{ij} = 1\right] = \frac{2}{j-i+1}$

Hence,

$$\mathbb{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2 \sum_{i=1}^{n-1} H_n \leq 2nH_n$$

I.e., $\mathbb{E}[C] = O(n \log n)$

Final push

Recall that

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

and $\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Hence,

$$\mathbb{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2 \sum_{i=1}^{n-1} H_n \leq 2nH_n$$

I.e., $\mathbb{E}[C] = O(n \log n)$

Final push

Recall that

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

and $\Pr\left[X_{ij} = 1\right] = \frac{2}{j-i+1}$

Hence,

$$\mathbb{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2 \sum_{i=1}^{n-1} H_n \leq 2nH_n$$

I.e., $\mathbb{E}[C] = O(n \log n)$

Final push

Recall that

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

and $\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Hence,

$$\mathbb{E}[C] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2\sum_{i=1}^{n-1} H_n \leq 2nH_n$$

I.e., $\mathbb{E}[C] = O(n \log n)$

using $k = j - i + 1$

Final push

Recall that

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

and $\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Hence,

$$\mathbb{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2 \sum_{i=1}^{n-1} H_n \leq 2nH_n$$

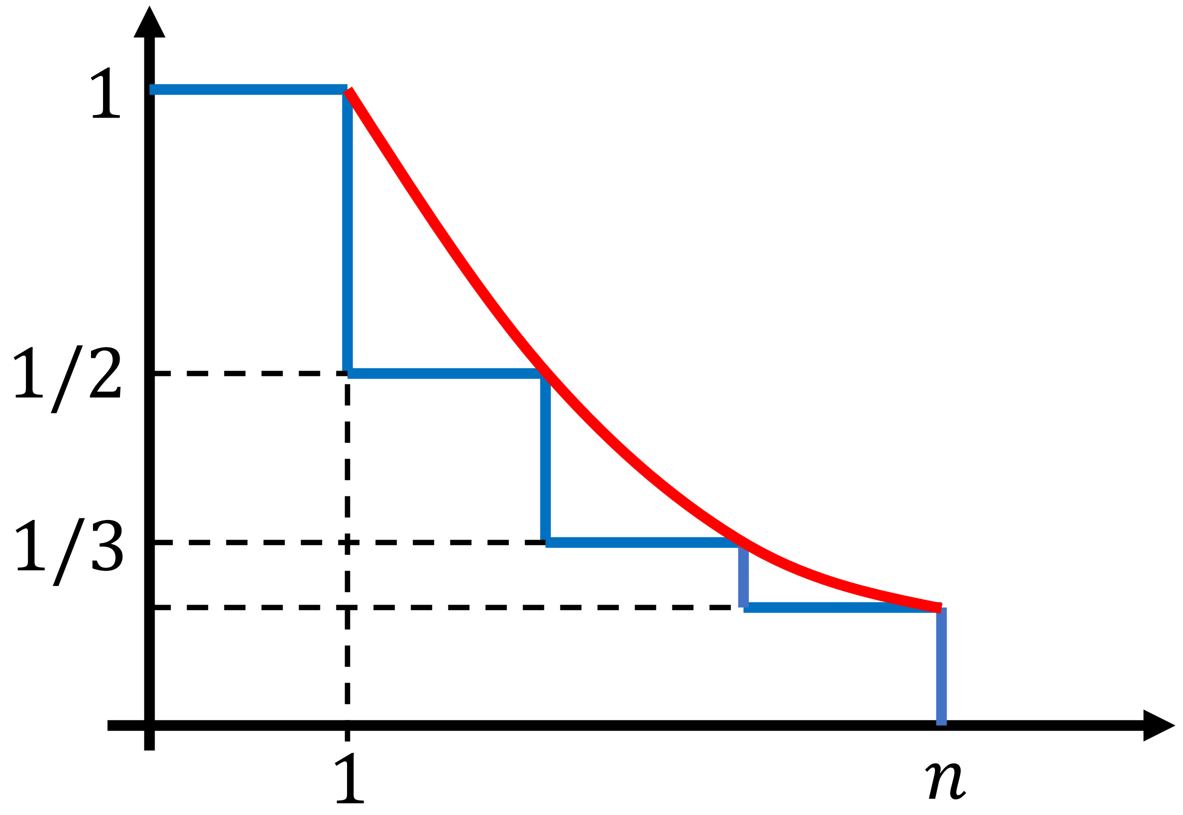
I.e., $\mathbb{E}[C] = O(n \log n)$

using $k = j - i + 1$

H_n is the n-the harmonic number

A remark on the Harmonic number

- A proof of $H_n \leq 1 + \ln n$



- H_n is the area below the blue line
- This is at most 1 + the area below the red line 1/x
- Hence, $H_n \leq 1 + \int_1^n \frac{\mathrm{d}x}{x} = 1 + \ln n$

Last slide

- Randomization: assumptions, benefits, and costs
- Examples of randomized algorithms (contention resolution, max-cut)
- Quicksort (naïve implementation, using the median)
- Randomized Quicksort (selecting the pivot at random)
- Analysis of Randomized Quicksort