Lazy Employee

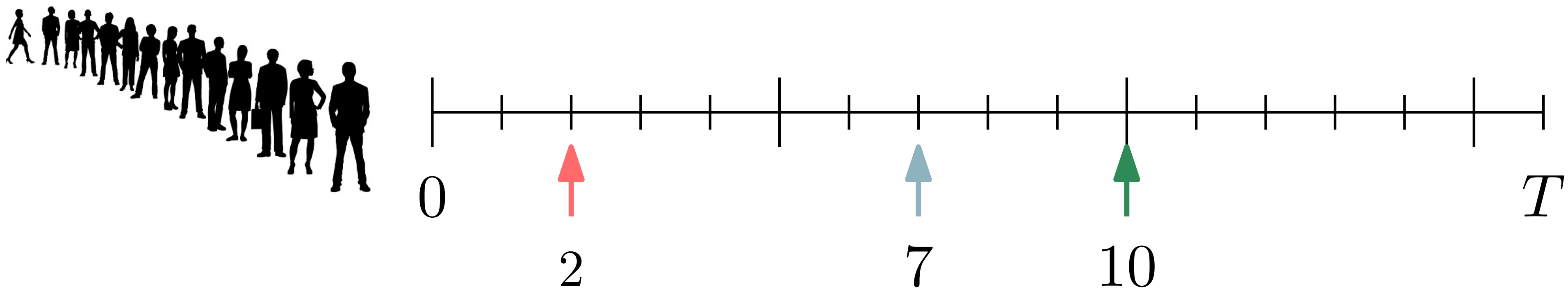
An office empoyee takes L minutes to serve a customer.
How slowly can he work?

- n customers arrive at distinct times $t_1, t_2, \ldots, t_n \in \mathbb{N}^+$
- All customers must leave by time T (closing time, $T > n$).
- The employee can only serve one customer at a time.
- The employee starts serving the next customer as soon as it finishes the current one. If no customer is available, he has to wait for one to arrive.

Goal: Maximize $L \in \mathbb{N}^+$.

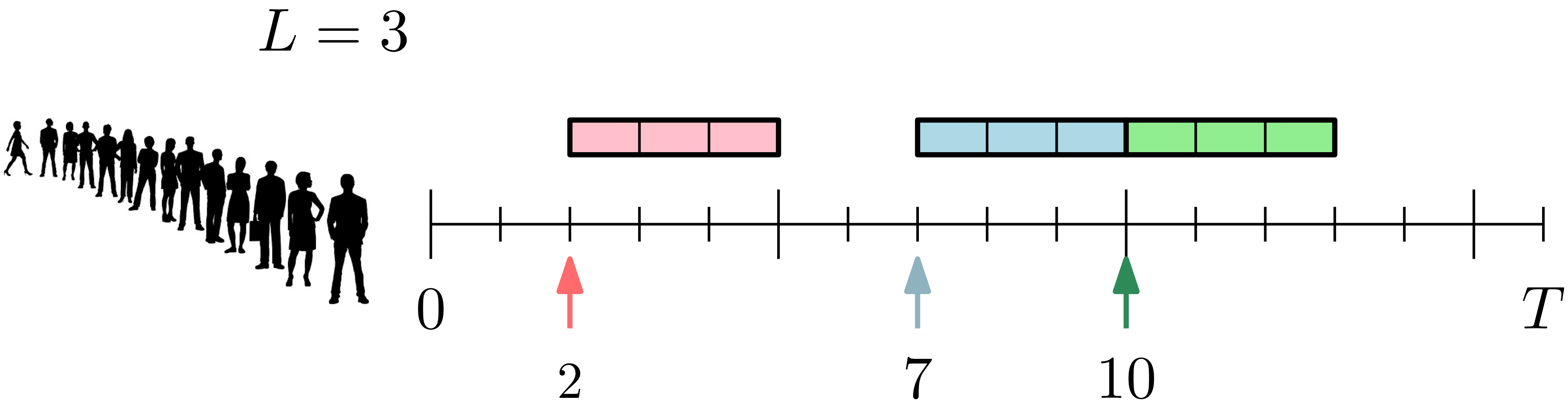
Example

- 3 customers arrive at times $t_1 = 2$, $t_2 = 10$, $t_3 = 7$
- All customers must leave by time $T = 16$



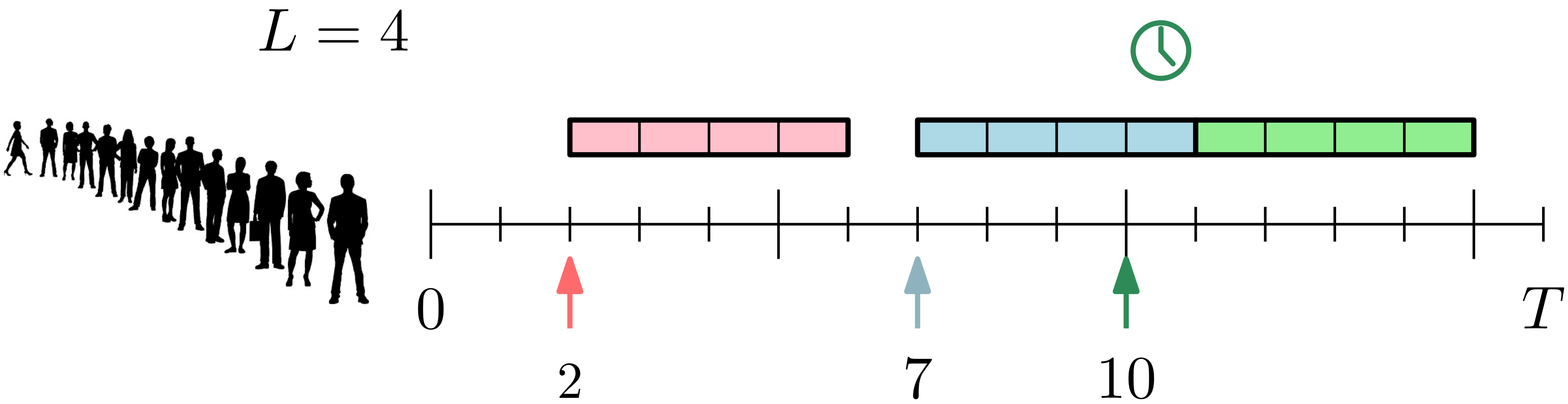
Example

- 3 customers arrive at times $t_1 = 2$, $t_2 = 10$, $t_3 = 7$
- All customers must leave by time $T = 16$



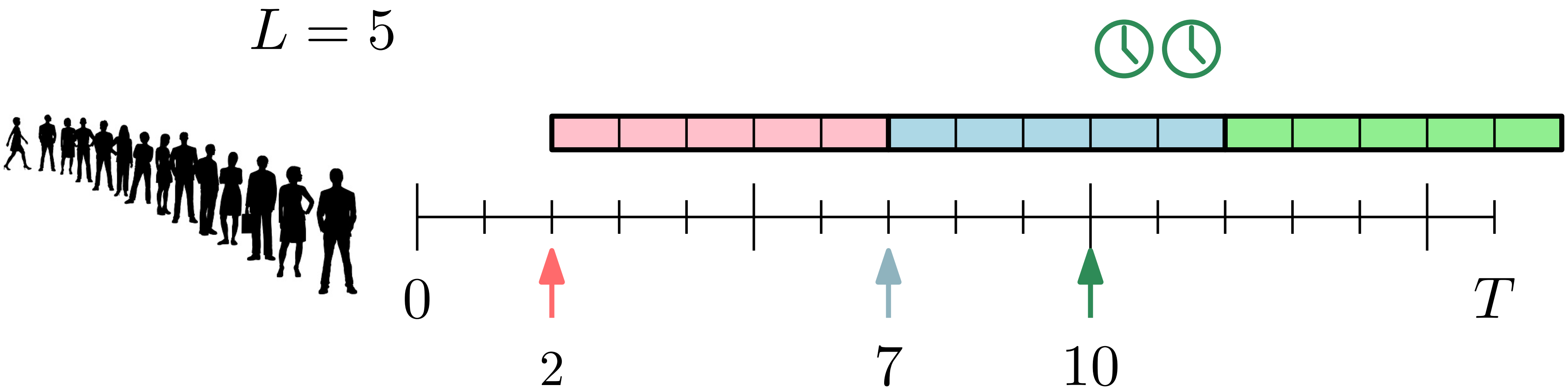
Example

- 3 customers arrive at times $t_1 = 2$, $t_2 = 10$, $t_3 = 7$
- All customers must leave by time $T = 16$



Example

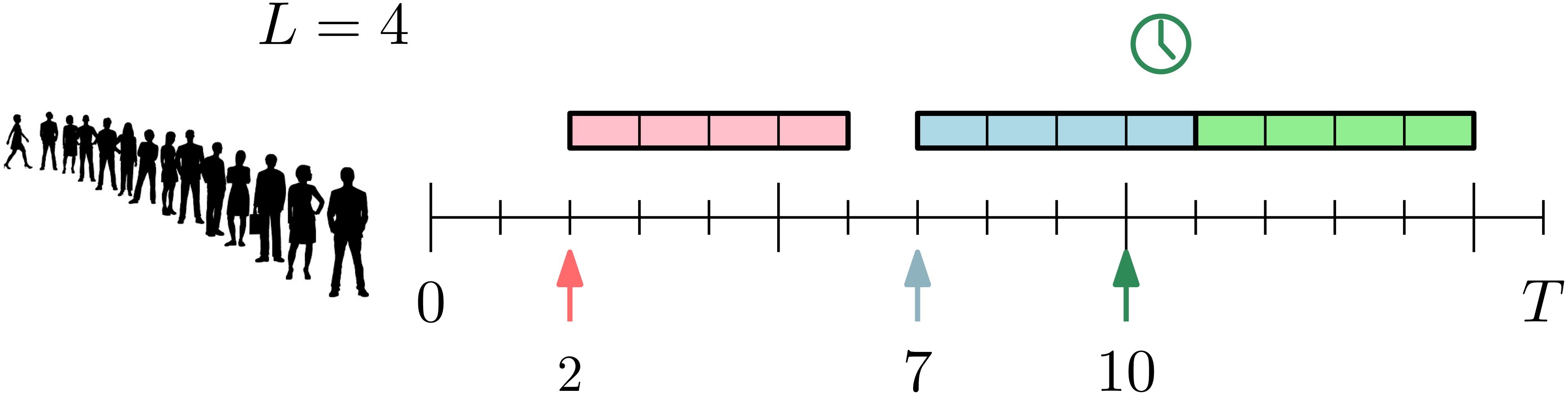
- 3 customers arrive at times $t_1 = 2$, $t_2 = 10$, $t_3 = 7$
- All customers must leave by time $T = 16$



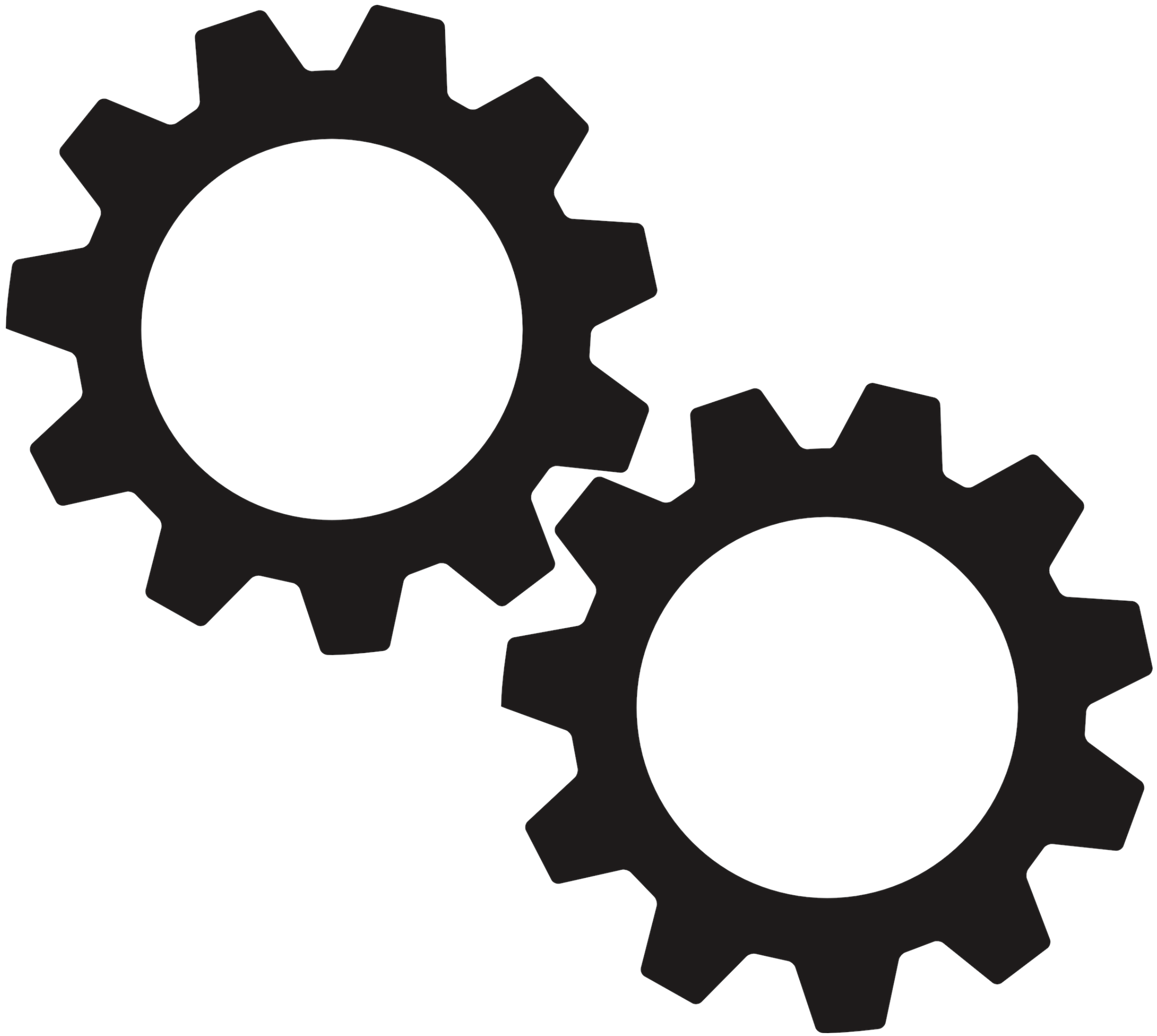
Example

- 3 customers arrive at times $t_1 = 2$, $t_2 = 10$, $t_3 = 7$
- All customers must leave by time $T = 16$

$L = 4$



Solution: $L = 4$



A simple solution

- For each possible value of $L = 2, \ldots, T$

  // Check whether all clients can be serverd

  - While there are unserved clients
    - Find and serve next client
- Stop at the first value of L for which clients can't be served
- Return $L - 1$

A simple solution

- For each possible value of $L = 2, \ldots, T$ $O(T)$

  // Check whether all clients can be serverd

  - While there are unserved clients $O(n)$
    - Find and serve next client $O(n)$
- Stop at the first value of L for which clients can't be served
- Return $L - 1$

Time complexity: $O(T \cdot n^2)$

A simple solution

- For each possible value of $L = 2, \ldots, \cancel{T}$ $\lceil T/n \rceil$ $O(T/n)$

  // Check whether all clients can be serverd

  - While there are unserved clients $O(n)$
    - Find and serve next client $O(n)$
- Stop at the first value of L for which clients can't be served
- Return $L - 1$

Time complexity: $O(T \cdot n)$

A simple solution

- **Preprocessing:** Sort $t_1, \ldots, t_n$ $O(n \log n)$
- For each possible value of $L = 2, \ldots, \cancel{T}$ $\lceil T/n \rceil$ $O(T/n)$

  // Check whether all clients can be serverd

  - While there are unserved clients $O(n)$
    - Find and serve next client $O(1)$
- Stop at the first value of L for which clients can't be served
- Return $L - 1$

Time complexity: $O(T + n \log n)$

A simple solution

- **Preprocessing:** Sort $t_1, \ldots, t_n$ $O(n \log n)$
- For each possible value of $L = 2, \ldots, \cancel{T}$ $\lceil T/n \rceil$ $O(T/n)$

  // Check whether all clients can be serverd

  - While there are unserved clients $O(n)$
    - Find and serve next client $O(1)$
- Stop at the first value of L for which clients can't be served
- Return $L - 1$

Trick/Technique: Sorting

Sorting can be a powerful preprocessing step.

A Possible Implementation

```
std::sort(arrival_times.begin(), arrival_times.end());

int L;
for(L=2; L<=1+T/n; L++)
{
    int time = 1; //Next available time
    for(const int t : arrival_times)
        time = std::max(time, t) + L;

    if(time > T)
        break;
}

std::cout << L-1 << "\n";
```

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$

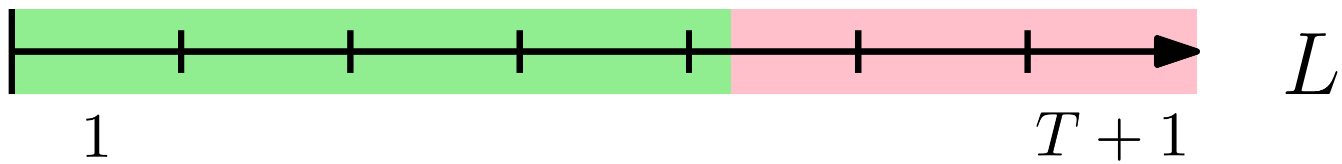
A Key Observation

Definition: We say that L is feasible if it allows to serve all customers by time T.

Observation: The property of being feasible is *monotone* w.r.t. L.

For $L > 1$, feasible$(L) \implies$ feasible$(L-1)$.

and $\neg$ feasible$(L-1) \implies \neg$ feasible(L).



A Key Observation

Definition: We say that L is feasible if it allows to serve all customers by time T.

Observation: The property of being feasible is *monotone* w.r.t. L.

Proof sketch

Suppose w.l.o.g. that the t_is are sorted. For a given L, let c_i^L the time after serving the i customers arriving at times $t_1, \dots, t_i$.

We prove by induction on $i = 1, \dots, n$, that $c_i^{L+1} > c_i^L$.

Base case $(i = 1)$:
$c_1^{L+1} = t_1 + L + 1 > t_1 + L = c_1^L$.

Induction step $(i > 1)$:
$c_i^{L+1} = \max\{c_{i-1}^{L+1}, t_i\} + L + 1 > \max\{c_{i-1}^L, t_i\} + L = c_i^L$. $\square$

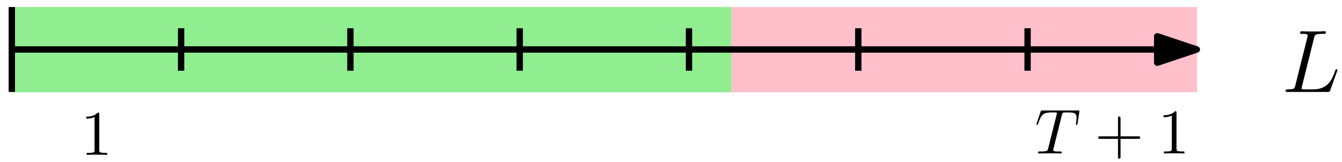
A Key Observation

Definition: We say that L is feasible if it allows to serve all customers by time T.

Observation: The property of being feasible is *monotone* w.r.t. L.

For $L > 1$, feasible$(L) \implies$ feasible$(L-1)$.

and $\neg$ feasible$(L-1) \implies \neg$ feasible(L).



Trick/Technique: Binary Search

Use **binary search** to efficiently find the largest feasible value of a monotone property.

A Possible Implementation

```cpp
//Returns the smallest index i in [min, max] such that
//p(i) is false.
//If no such index exists, returns max.
template<typename T>
    T binary_search(T min, T max, bool (*p)(T) )
{
    while(min < max)
    {
        if(T mid = (min+max)/2; p(mid))
            min = mid+1;
        else
            max = mid;
    }

    return max;
}
```

Time complexity: $O(\log(\text{max} - \text{min}))$

A Possible Implementation

```
bool feasible(int L)
{
    int time=1;
    for(const int t : arrival_times)
        time = std::max(time, t) + L;

    return time<=T;
}

std::sort(arrival_times.begin(), arrival_times.end());
std::cout << binary_search(2, T, feasible)-1 << "\n";
```

A Possible Implementation

```
bool feasible(int L)
{
    int time=1;
    for(const int t : arrival_times)
        time = std::max(time, t) + L;

    return time<=T;
}

std::sort(arrival_times.begin(), arrival_times.end());
std::cout << binary_search(2, T, feasible)-1 << "\n";
```

Time complexity: $O(n \log T)$

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$
+ Binary Search for L	$O(n \log T)$

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$
+ Binary Search for L	$O(n \log T)$

$O(n \log T)$ is never worse than $O(T + n \log n)$

Solutions so far

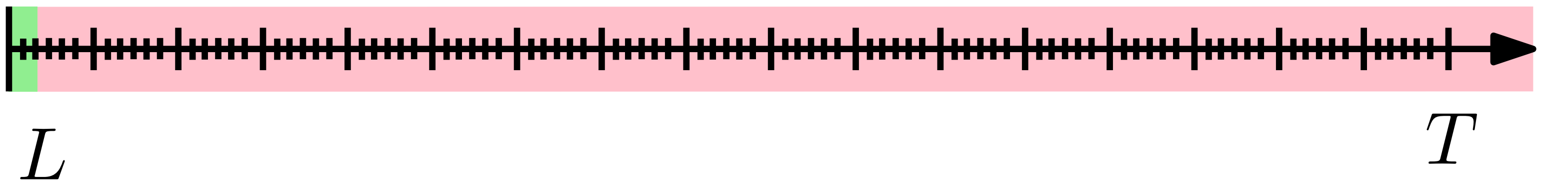
Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$
+ Binary Search for L	$O(n \log T)$

$O(n \log T)$ is never worse than $O(T + n \log n)$

What if T is large (e.g., 2^n) but L is small?

Handling large values of L

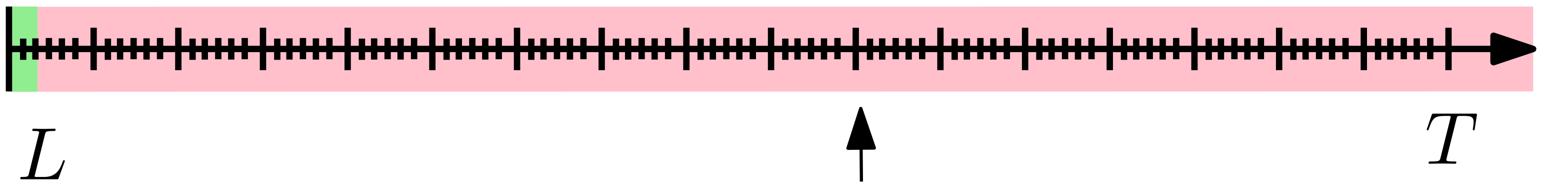
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

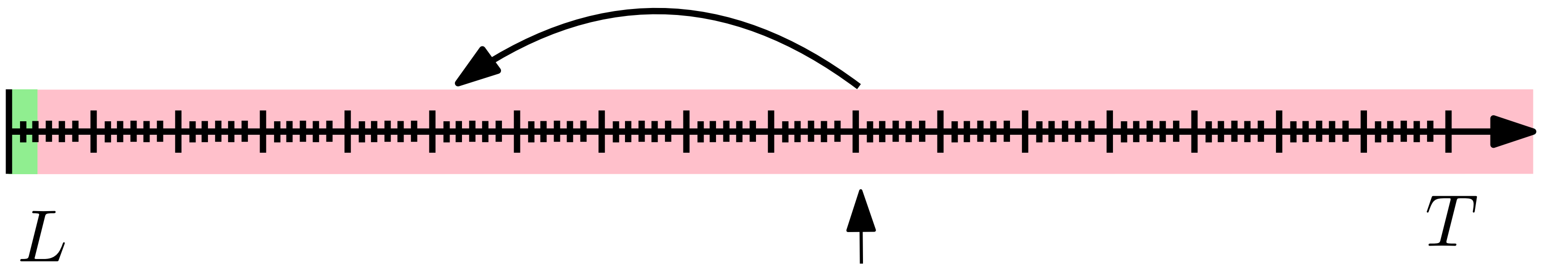
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

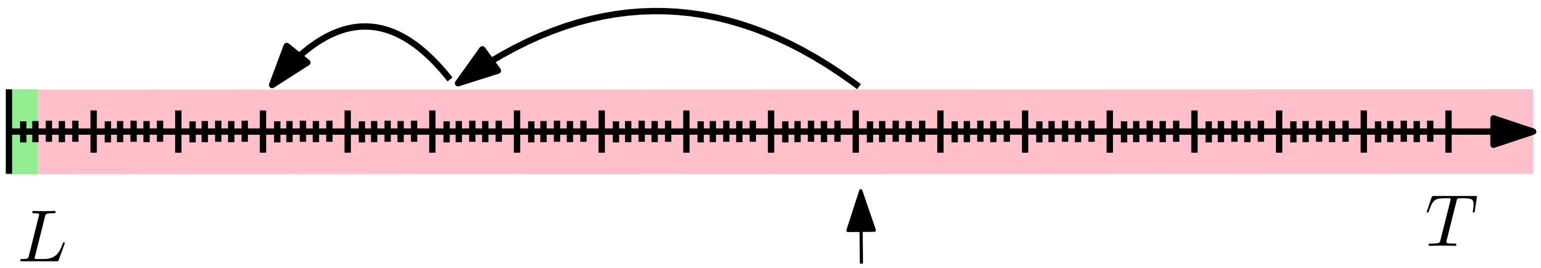
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

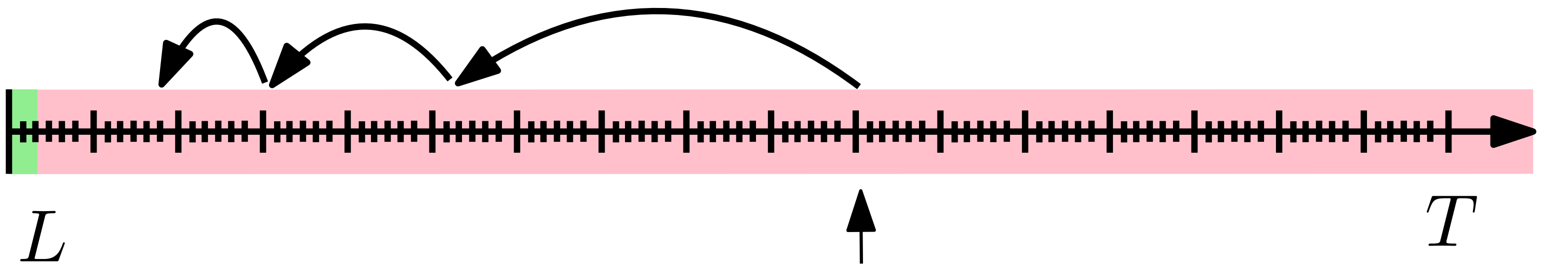
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

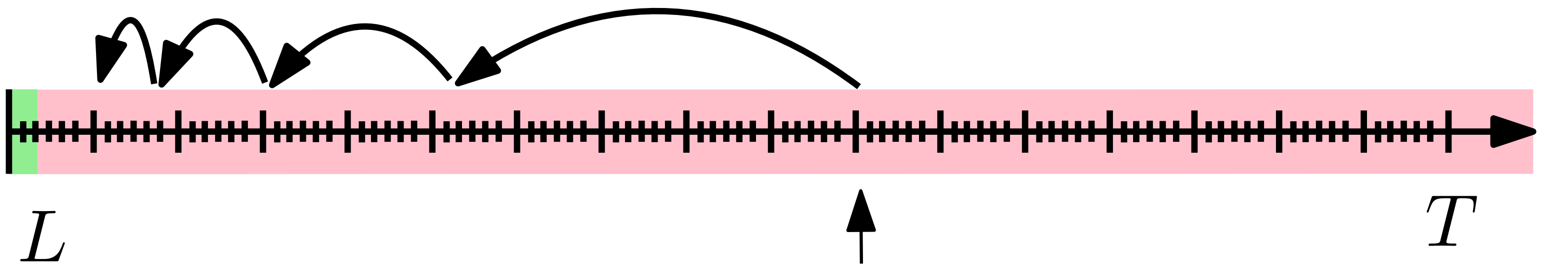
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

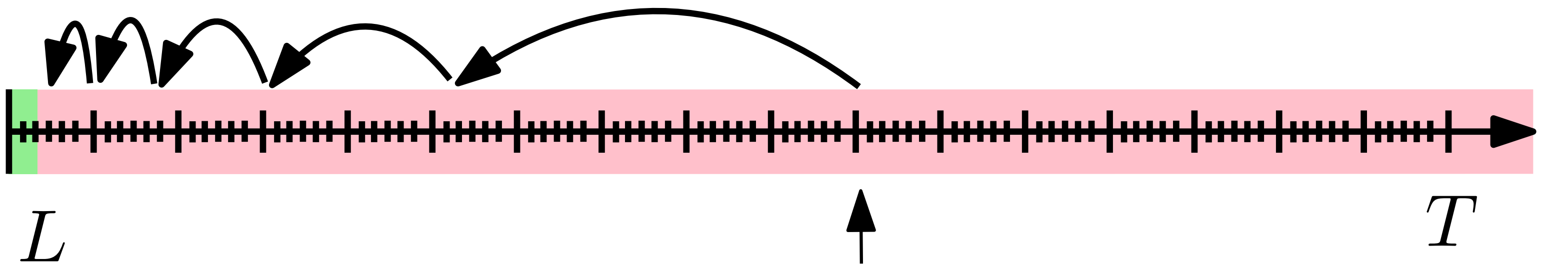
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

What if T is large (e.g., 2^n) but L is small?

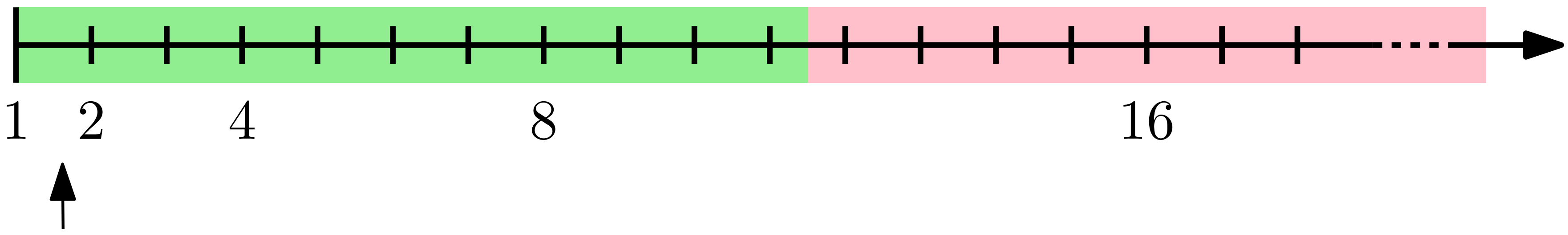


(not to scale)

Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

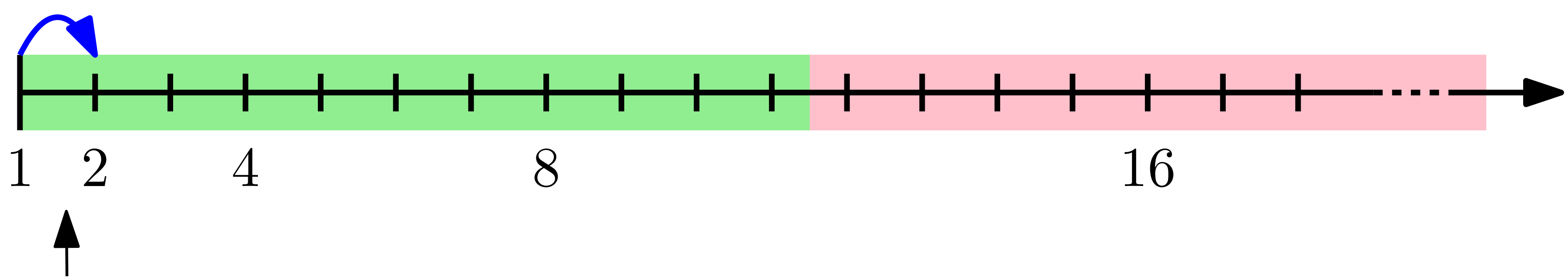
$\implies U \in (L, 2L]$.



Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

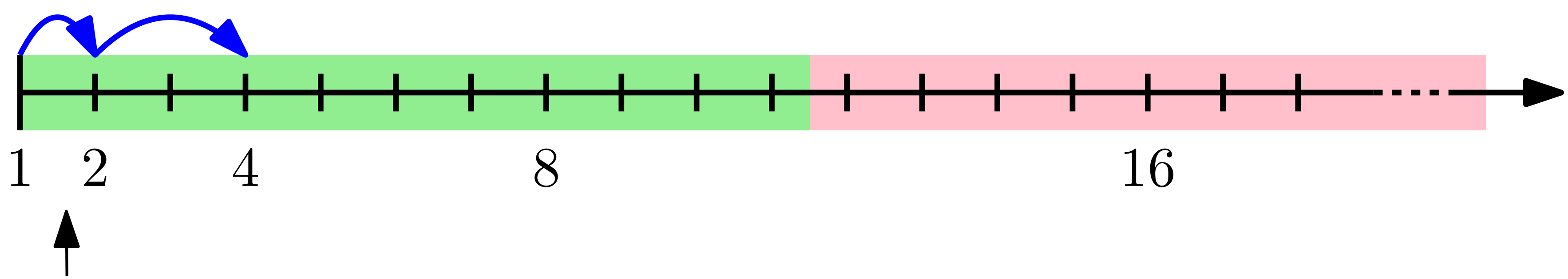
$$\implies U \in (L, 2L].$$



Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

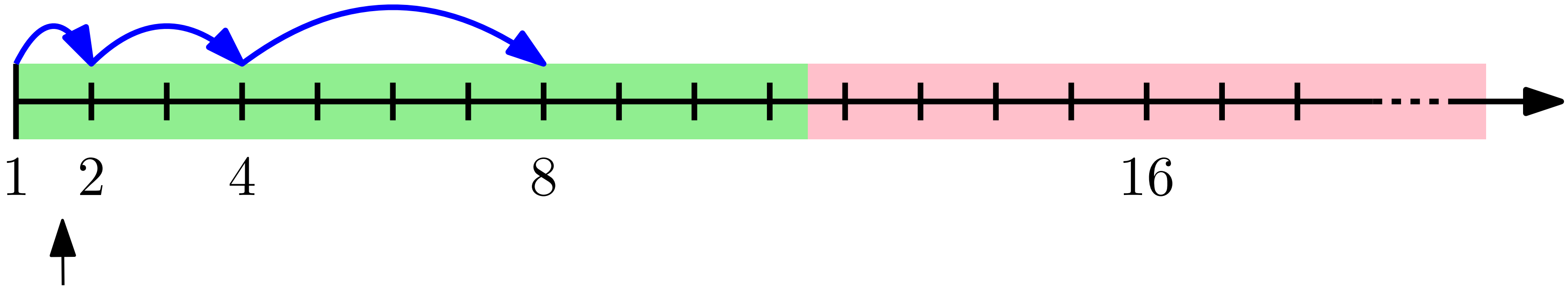
$\implies U \in (L, 2L]$.



Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

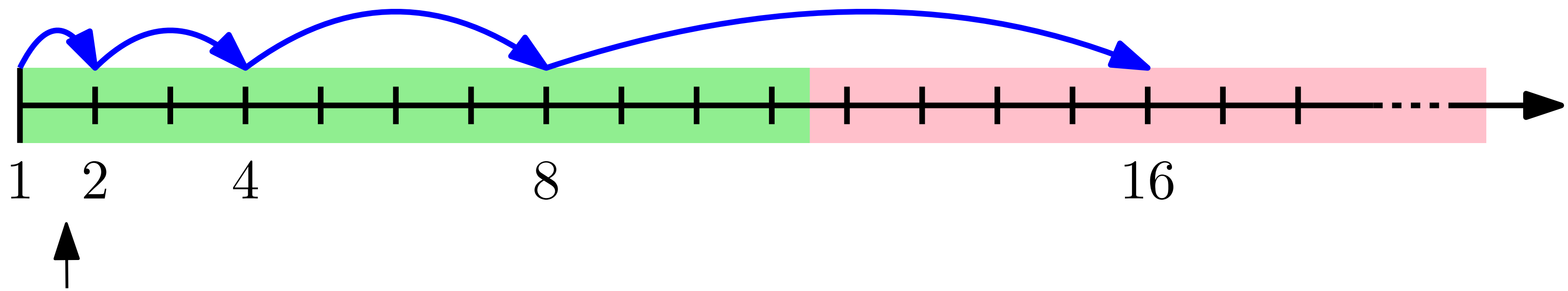
$$\implies U \in (L, 2L].$$



Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

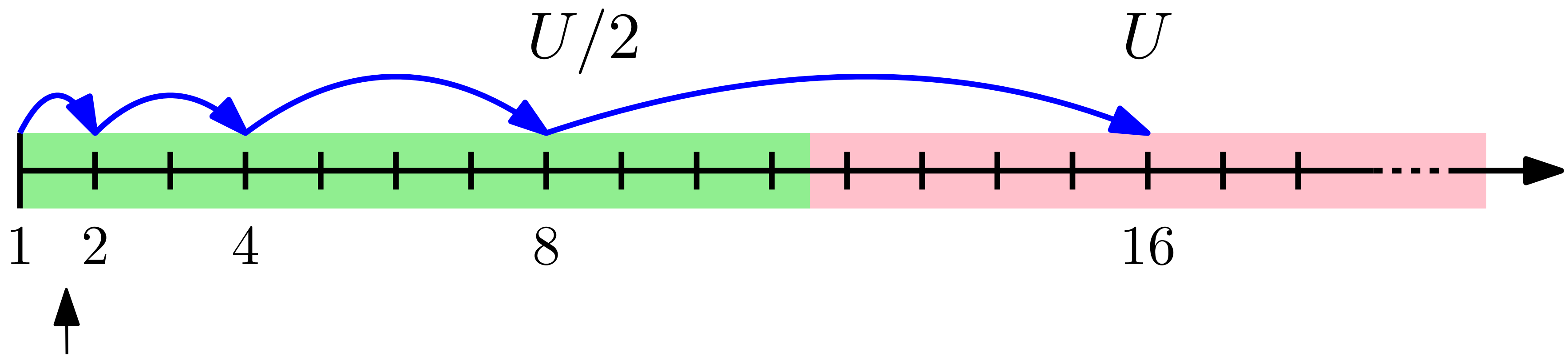
$\implies U \in (L, 2L]$.



Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

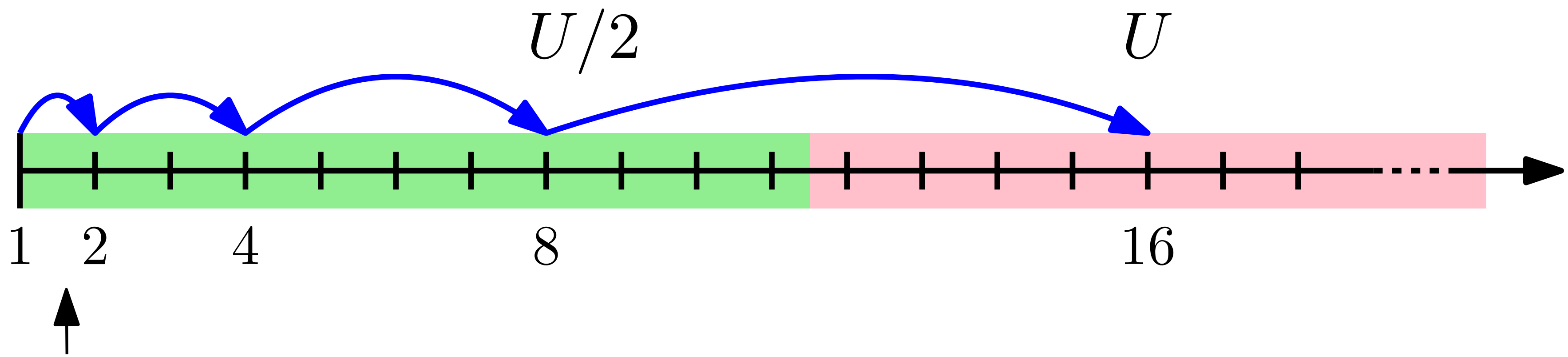
$\implies U \in (L, 2L]$.



Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

$$\implies U \in (L, 2L].$$



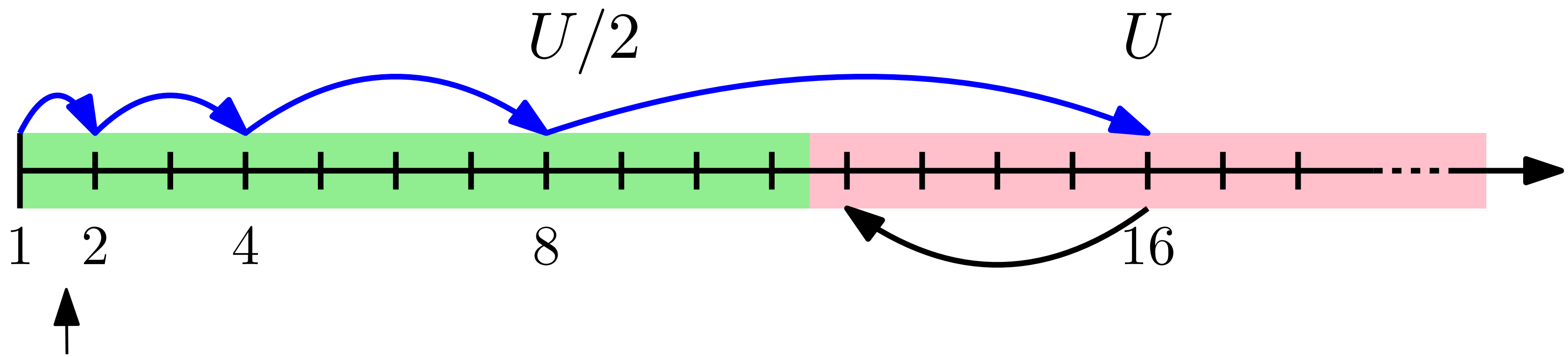
Trick/Technique: Exponential Search

Use **exponential search** to efficiently find an upper bound for applying **binary search**.

Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

$$\implies U \in (L, 2L].$$



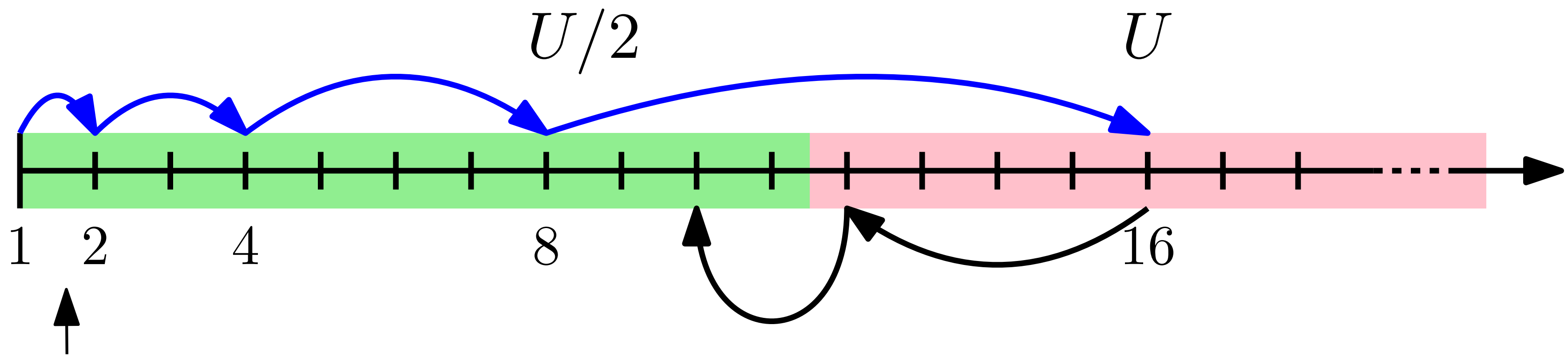
Trick/Technique: Exponential Search

Use **exponential search** to efficiently find an upper bound for applying **binary search**.

Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

$$\implies U \in (L, 2L].$$



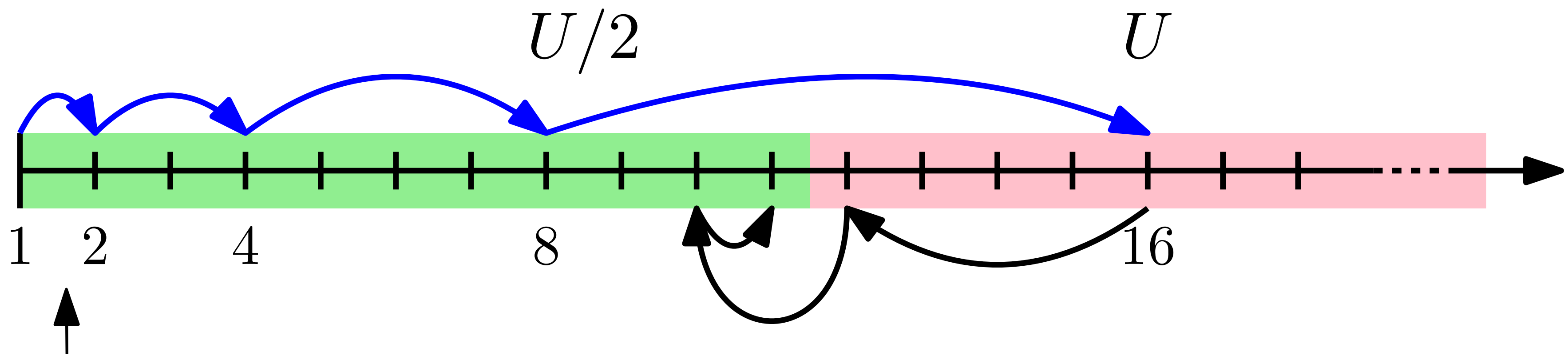
Trick/Technique: Exponential Search

Use **exponential search** to efficiently find an upper bound for applying **binary search**.

Exponential Search

Idea: Check $U = 1, 2, 4, 8, \ldots,$ until feasible(U) is false.

$$\implies U \in (L, 2L].$$



Trick/Technique: Exponential Search

Use **exponential search** to efficiently find an upper bound for applying **binary search**.

A Possible Implementation

```
//If p(min) is false returns min
//Otherwise returns an index min+i in (min, max] such that
//p(min+i) is false and p(min+i/2) is true.
//If no such index exists, returns max.
template<typename T>
       T exponential_search(T min, T max, bool (*p)(T) )
{
    if(!p(min))
        return min;

    for(T i=1; min+i<max; i*=2)
        if(!p(min+i))
            return min+i;

    return max;
}
```

Time complexity: $O(\log(\mathsf{max} - \mathsf{min}))$

A Possible Implementation

```cpp
std::sort(arrival_times.begin(), arrival_times.end());
int U = exponential_search(2, T, feasible);
std::cout << binary_search(U/2, U, feasible) - 1;
```

Time complexity: $O(n \log n + n \log L)$

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$
+ Binary Search for L	$O(n \log T)$

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$
+ Binary Search for L	$O(n \log T)$
+ Exponential search	$O(n \log n + n \log L)$

Solutions so far

Naive algorithm	$O(T \cdot n^2)$
Naive algorithm (better analysis)	$O(T \cdot n)$
+ Preprocessing (sort arrival times)	$O(T + n \log n)$
+ Binary Search for L	$O(n \log T)$
+ Exponential search	$O(n \log n + n \log L)$

$O(n \log n + n \log L)$ is never worse than $O(n \log T)$

Trick/Technique: Sorting

Sorting can be a powerful preprocessing step.

Trick/Technique: Sorting

Sorting can be a powerful preprocessing step.

Trick/Technique: Binary Search

Use **binary search** to efficiently find the largest feasible value of a monotone property.

Trick/Technique: Sorting

Sorting can be a powerful preprocessing step.

Trick/Technique: Binary Search

Use **binary search** to efficiently find the largest feasible value of a monotone property.

Trick/Technique: Exponential Search

Use **exponential search** to efficiently find an upper bound for applying **binary search**.