Lazy Employee

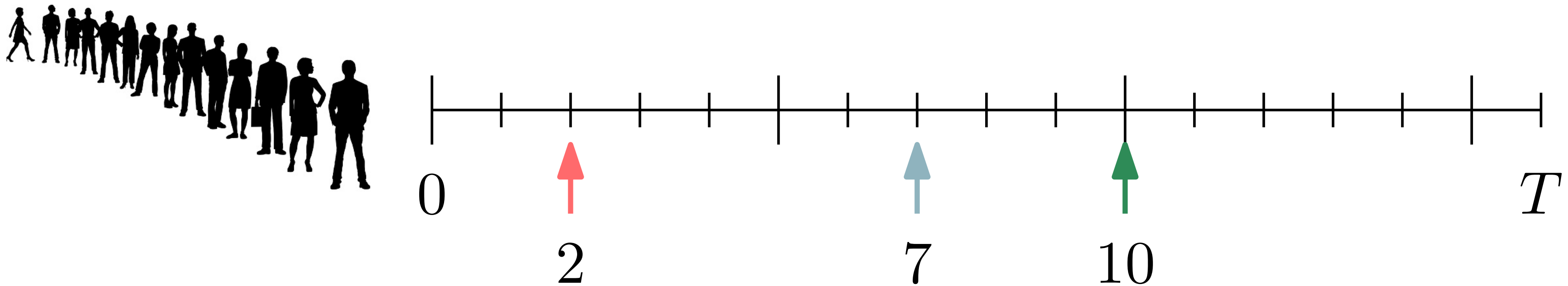
An office empoyee takes L minutes to serve a customer.
How slowly can he work?

- n customers arrive at times $t_1, t_2, \ldots, t_n \in \mathbb{N}^+$
- The last customer must leave by time T (closing time).
- 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

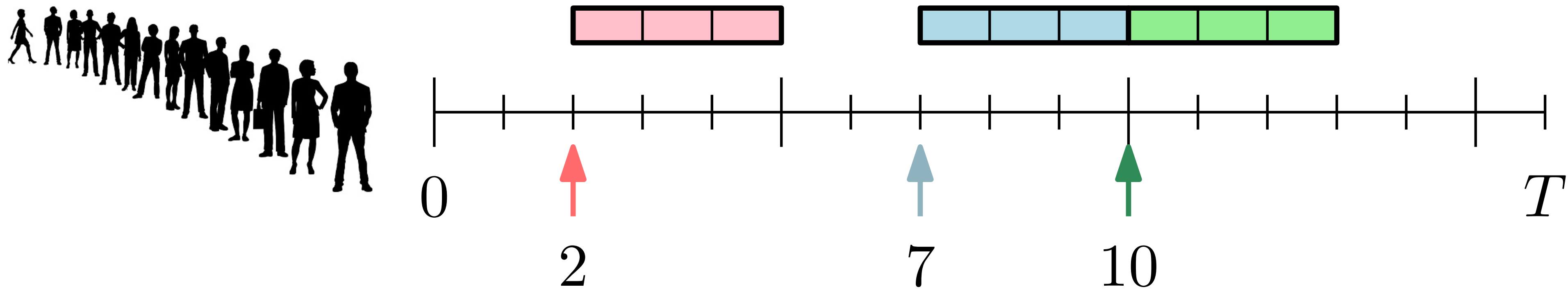
- 3 customers arrive at times $t_1 = 2$, $t_2 = 7$, $t_3 = 10$
- Last customer must leave by time $T = 16$



Example

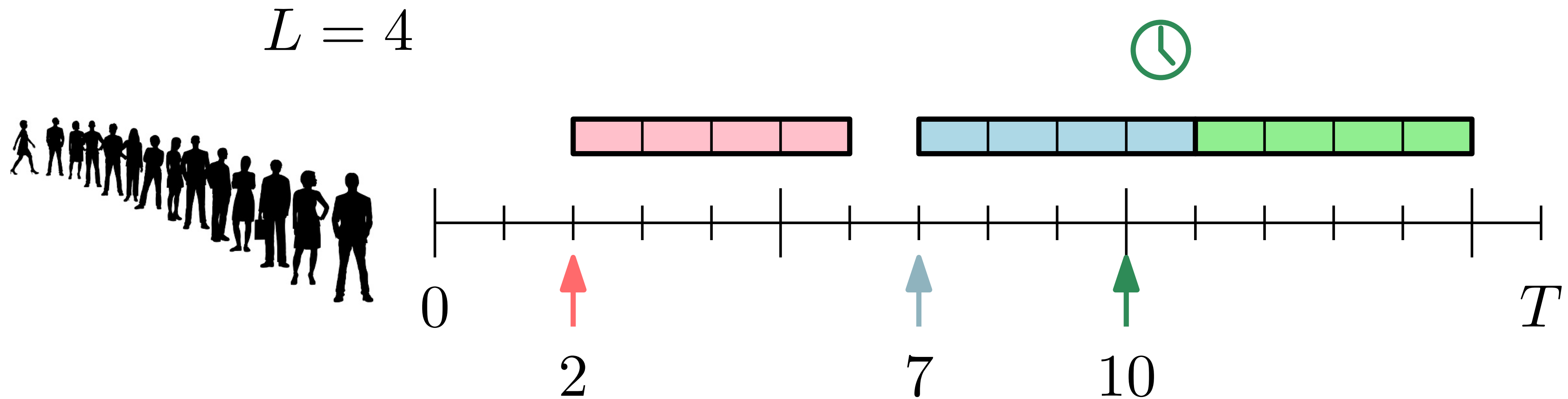
- 3 customers arrive at times $t_1 = 2$, $t_2 = 7$, $t_3 = 10$
- Last customer must leave by time $T = 16$

$L = 3$



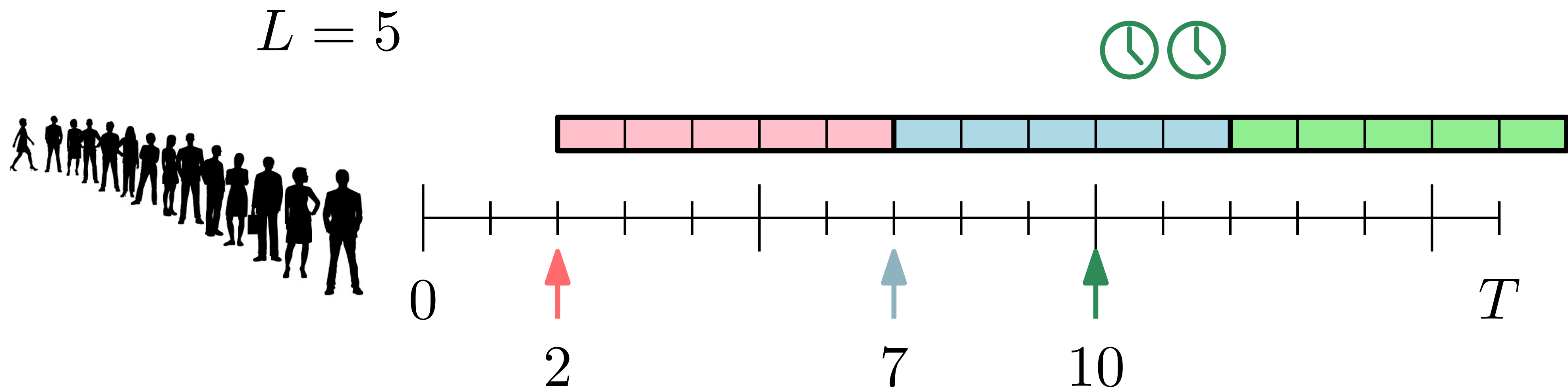
Example

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



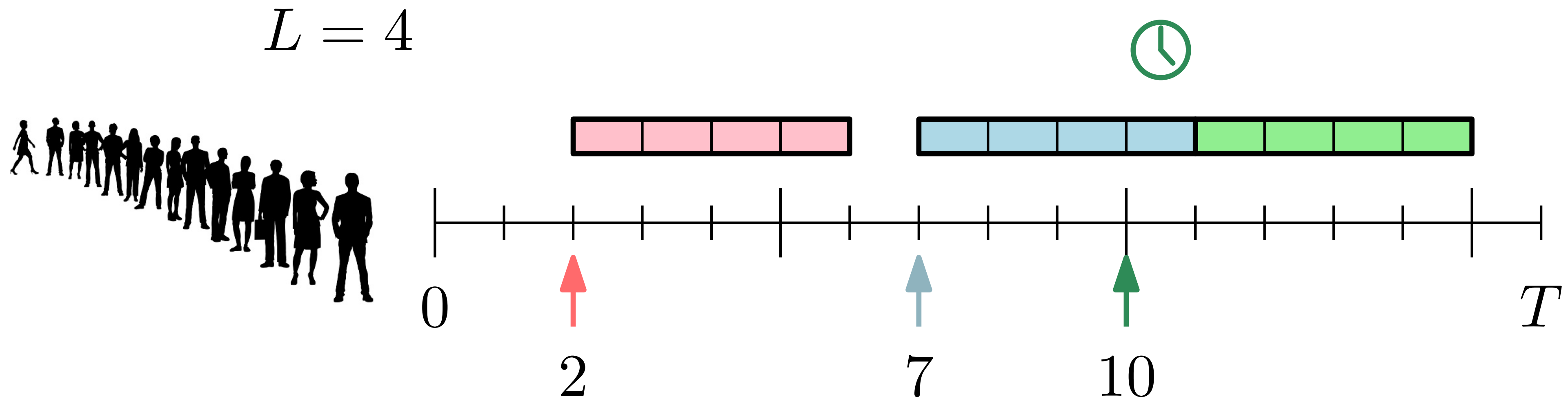
Example

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

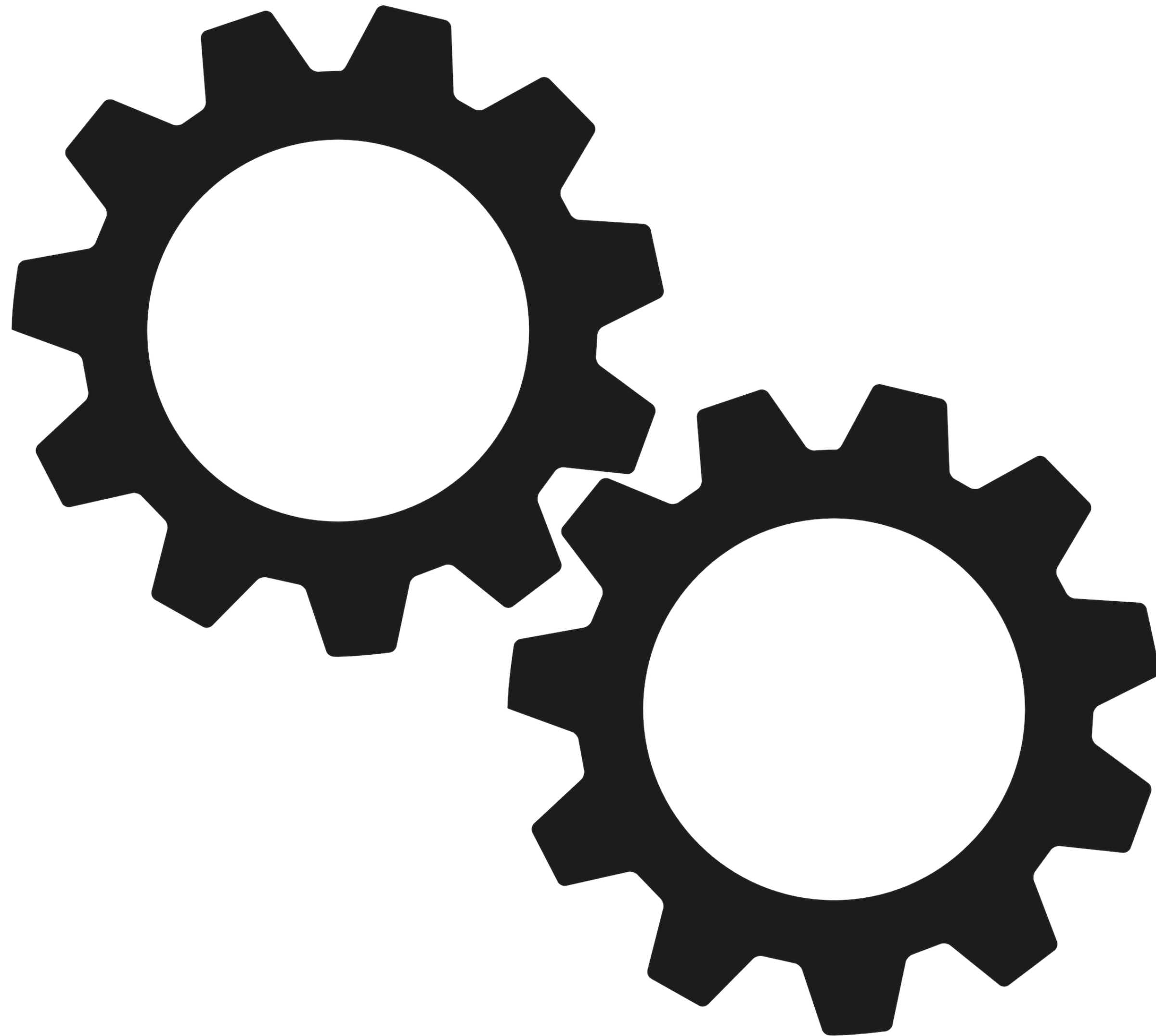


Example

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



Solution: $L = 4$



A simple solution

- For each possible value of $L = 1, \ldots, T+1$

  // 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

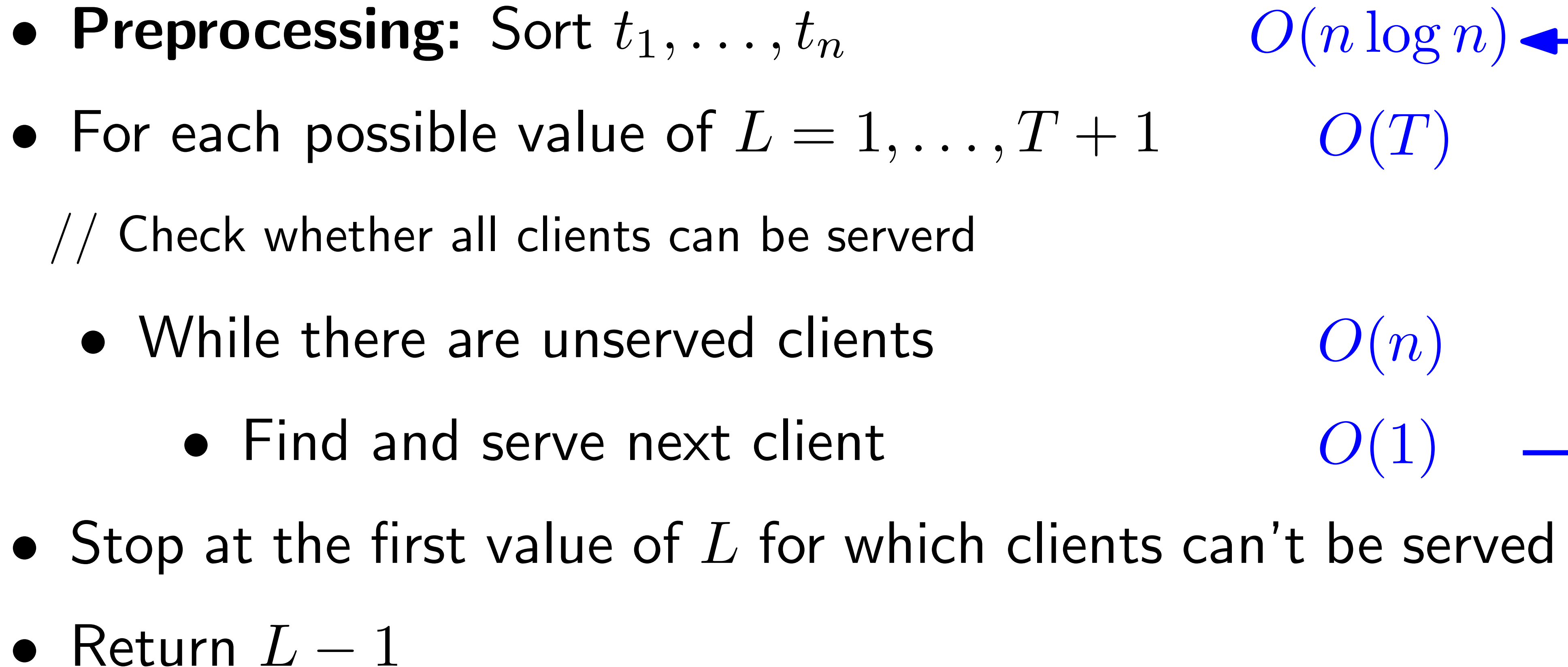
- For each possible value of $L = 1, \ldots, T+1$ $\quad O(T)$

  // Check whether all clients can be serverd

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

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

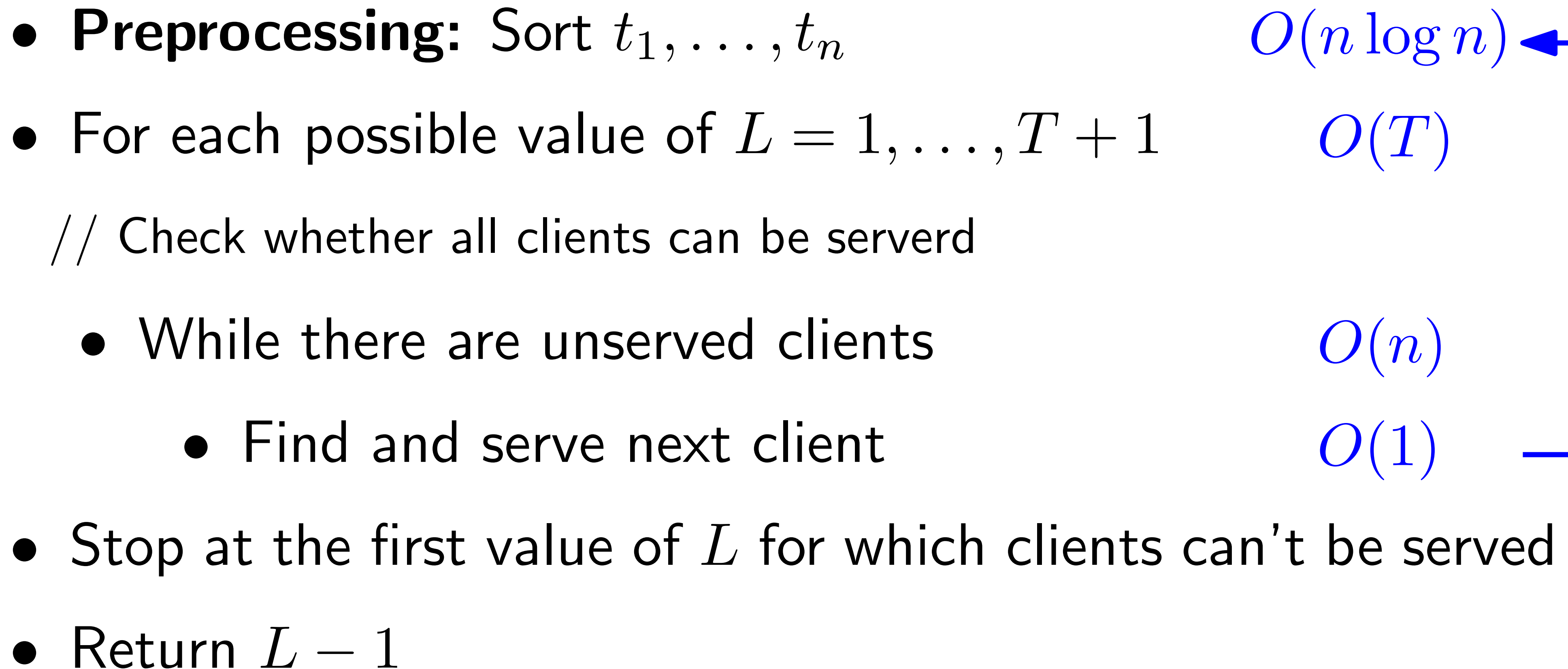
A simple solution



Time complexity: $O(Tn + n \log n) = O(Tn)$

(since we can assume $T \geq n$)

A simple solution



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<=T+1; 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";
```

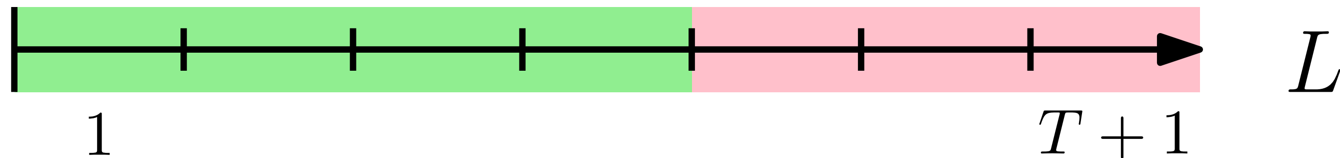
A Key Observation

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. Let c_i^T the time after serving the i customres arriving at times $t_1, \ldots, t_i$.

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

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

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

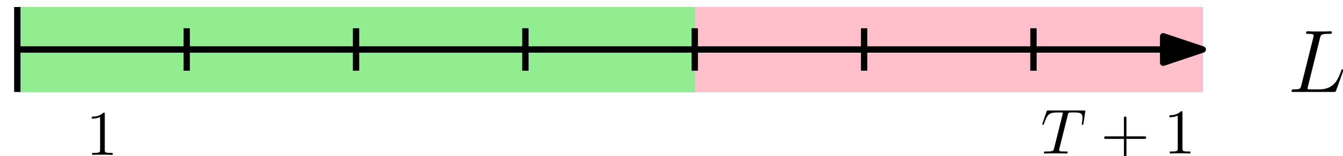
A Key Observation

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

```
//Returns the largest index i in [min, max] such that
//p(i) is true.
//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 min;
}
```

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(1, T, feasible) << "\n";
```

A Possible Implementation

```cpp
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(1, T, feasible) << "\n";
```

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

A Possible Implementation

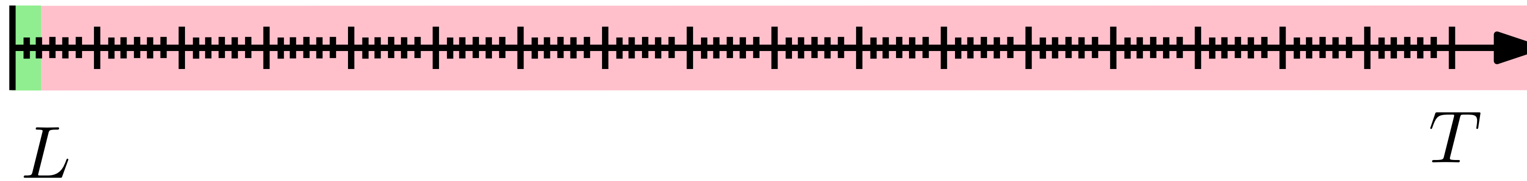
```
bool feasible(int L)
{
    int time=1;
    for(const int t : arrival_times)
        time = std::max(time, t) + L;

    return time<=T;
}

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

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

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

Handling large values of L

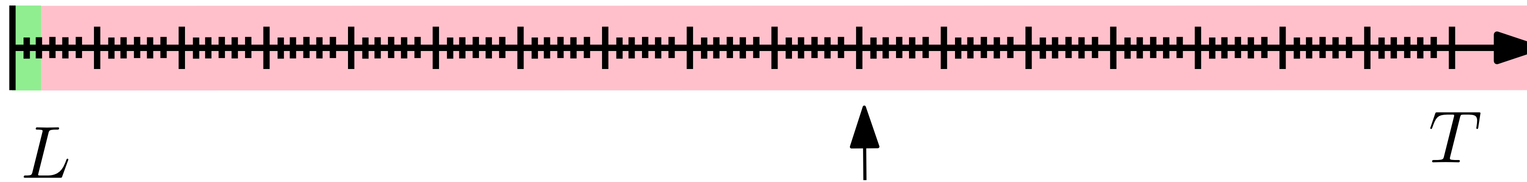
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

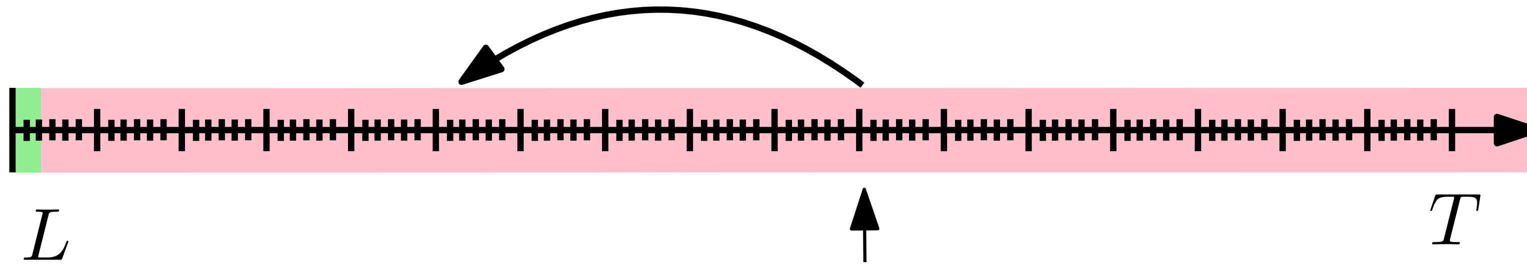
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

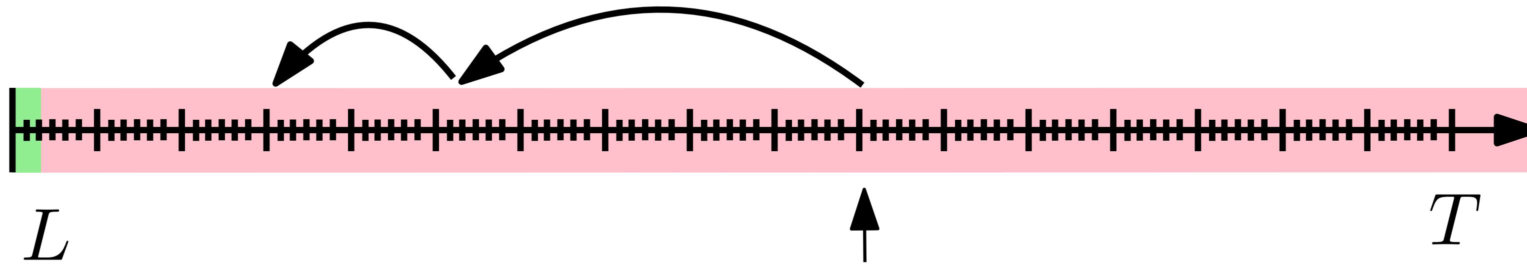
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

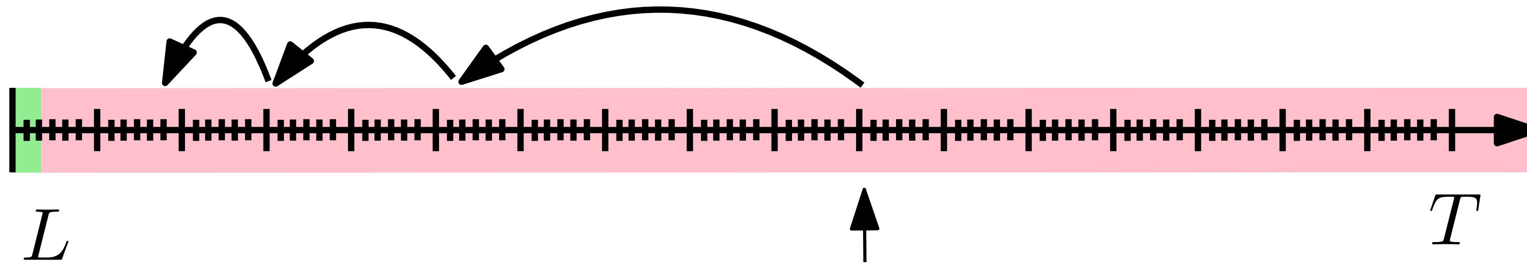
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

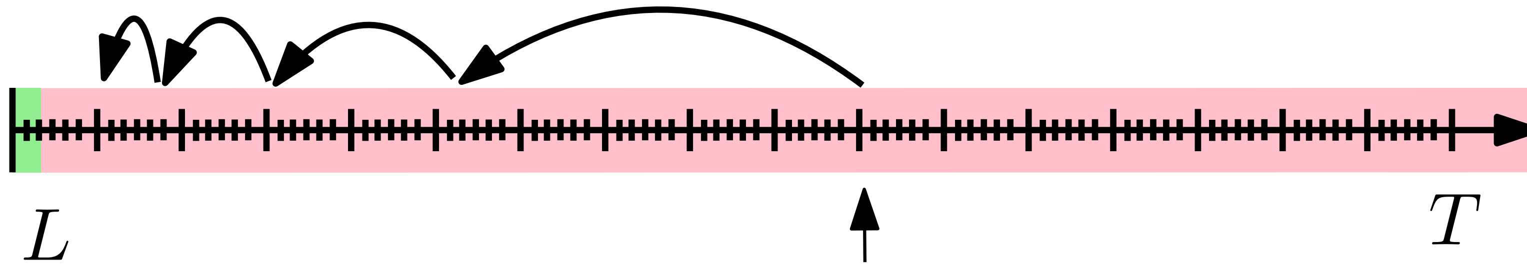
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

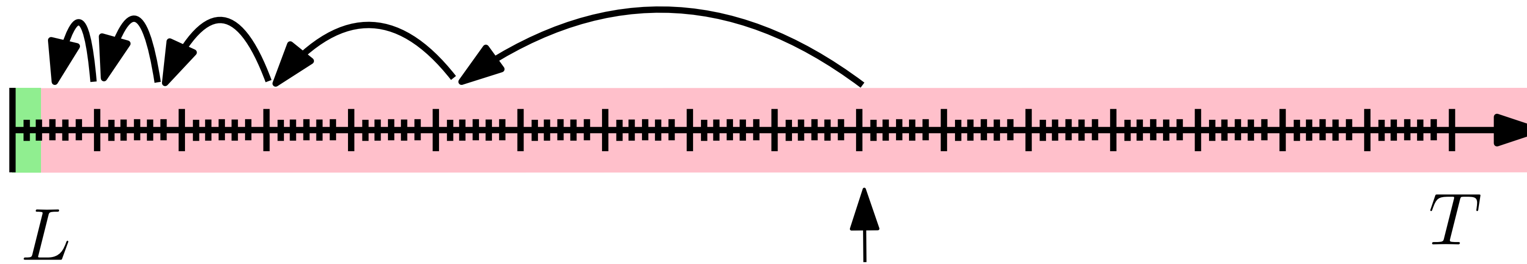
What if T is large (e.g., 2^n) but L is small?



(not to scale)

Handling large values of L

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

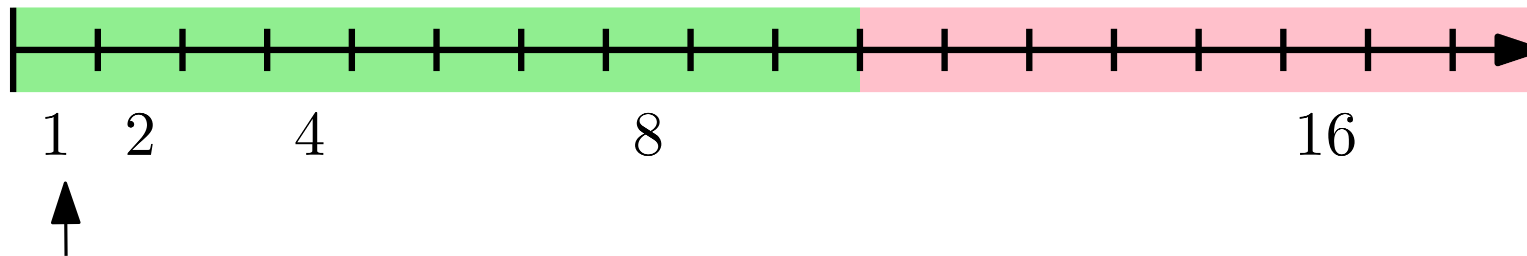


(not to scale)

Exponential Search

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

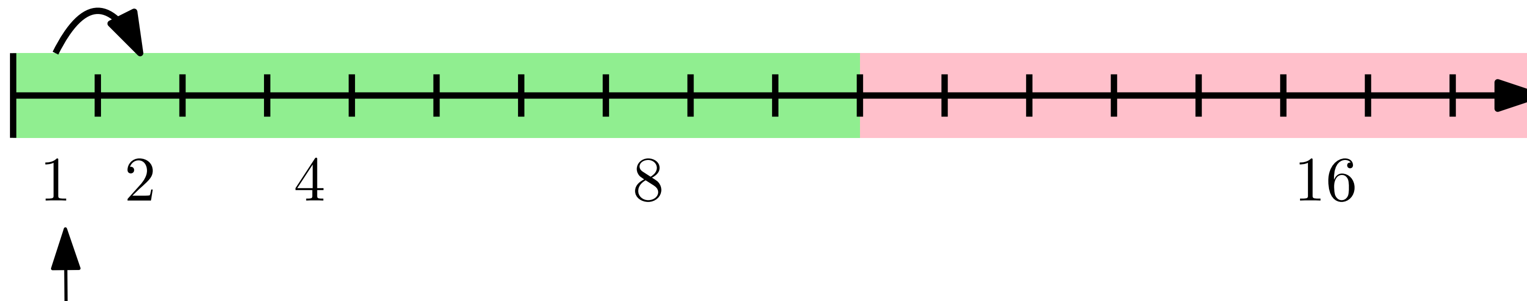
$\implies U \in (L, 2L]$.



Exponential Search

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

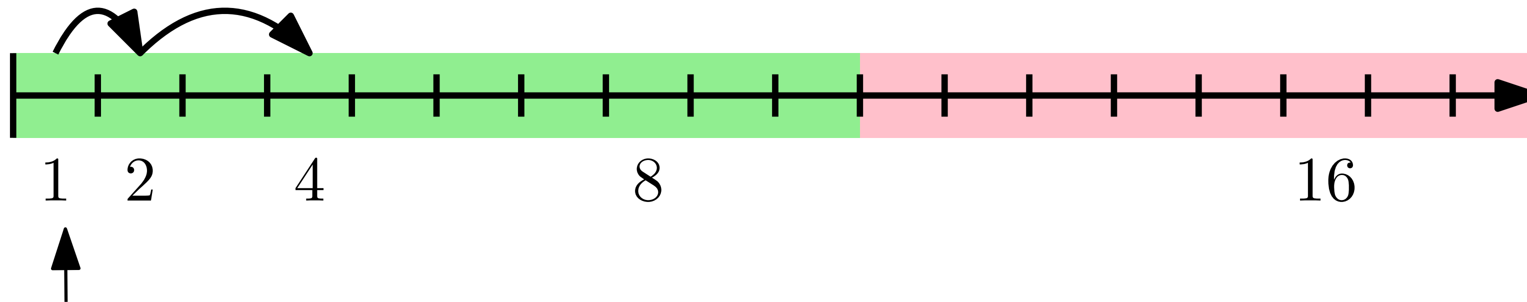
$\implies U \in (L, 2L]$.



Exponential Search

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

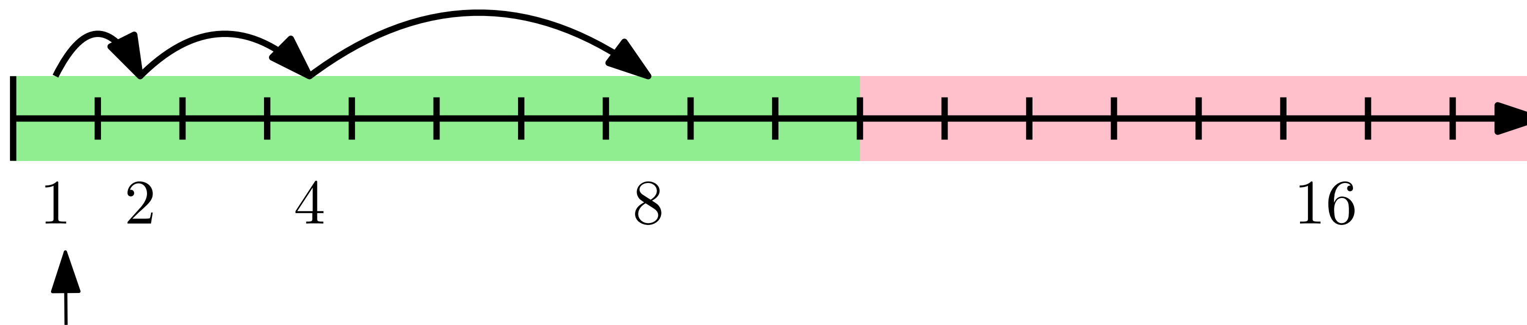
$\implies U \in (L, 2L]$.



Exponential Search

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

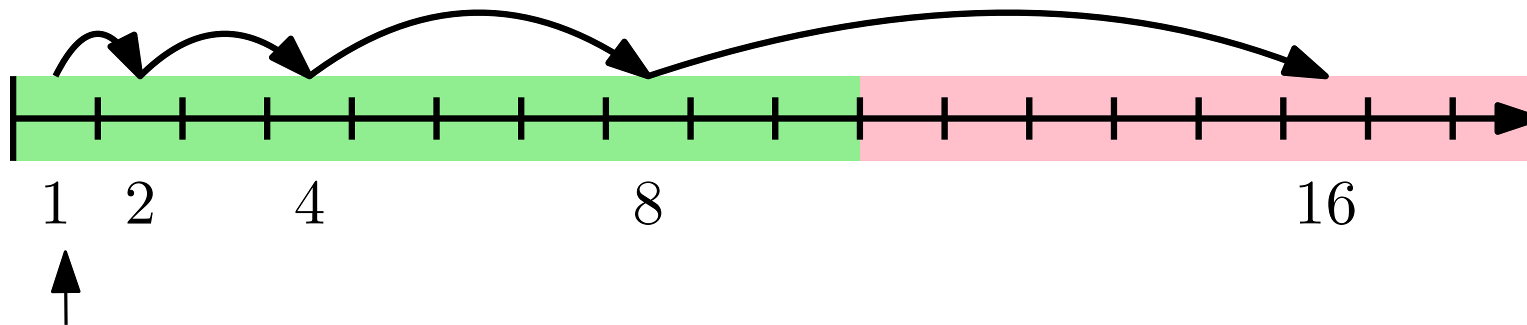
$\implies U \in (L, 2L]$.



Exponential Search

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

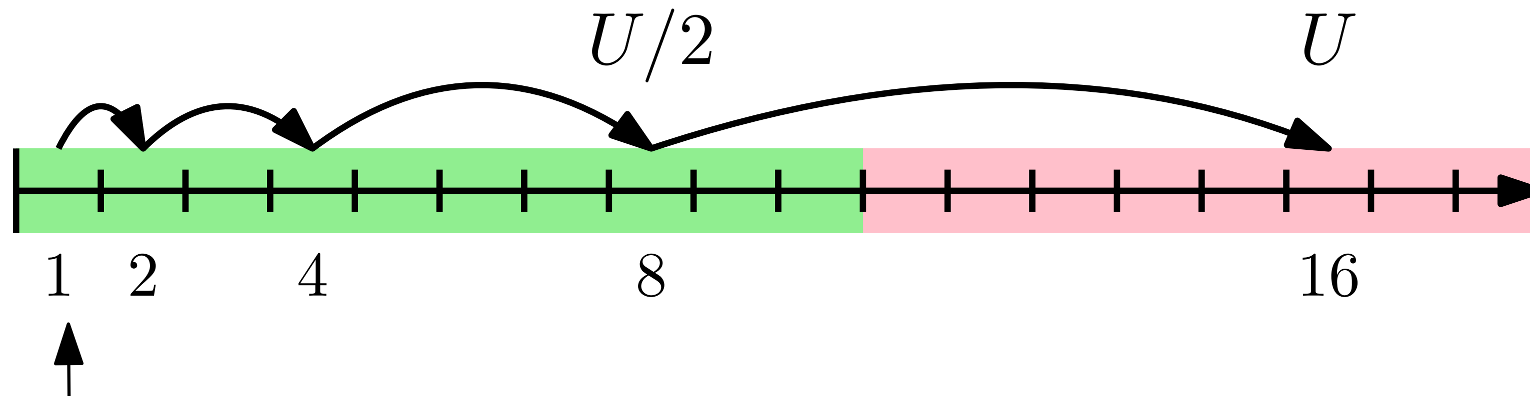
$$\implies U \in (L, 2L].$$



Exponential Search

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

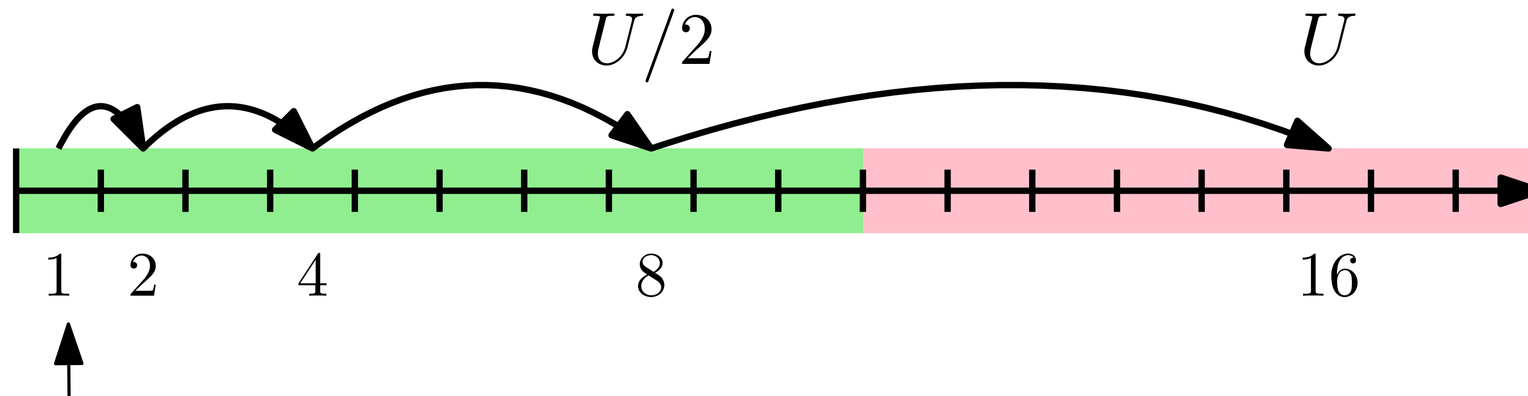
$\implies U \in (L, 2L]$.



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.

A Possible Implementation

```
//If p(min) is false returns p(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: $O(\log(\text{max} - \text{min}))$

A Possible Implementation

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

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

(instead of $O(n \log T)$)

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.