Analyzing algorithms & Divide-and-conquer

Yan Gu

Course announcement

- **HW 1 due this Friday**
  - Try to start working on it soon if you haven't
  - Come to the office hours if you need help

- **Course policy test: due next Tuesday**
  - 1 point to your final grade, and required
  - Resubmitable multiple-choices problems

- **Regarding course-related logistics**
  - Contact Zijin for non-homework-related questions
  - Contact Xiangyun for homework-related questions

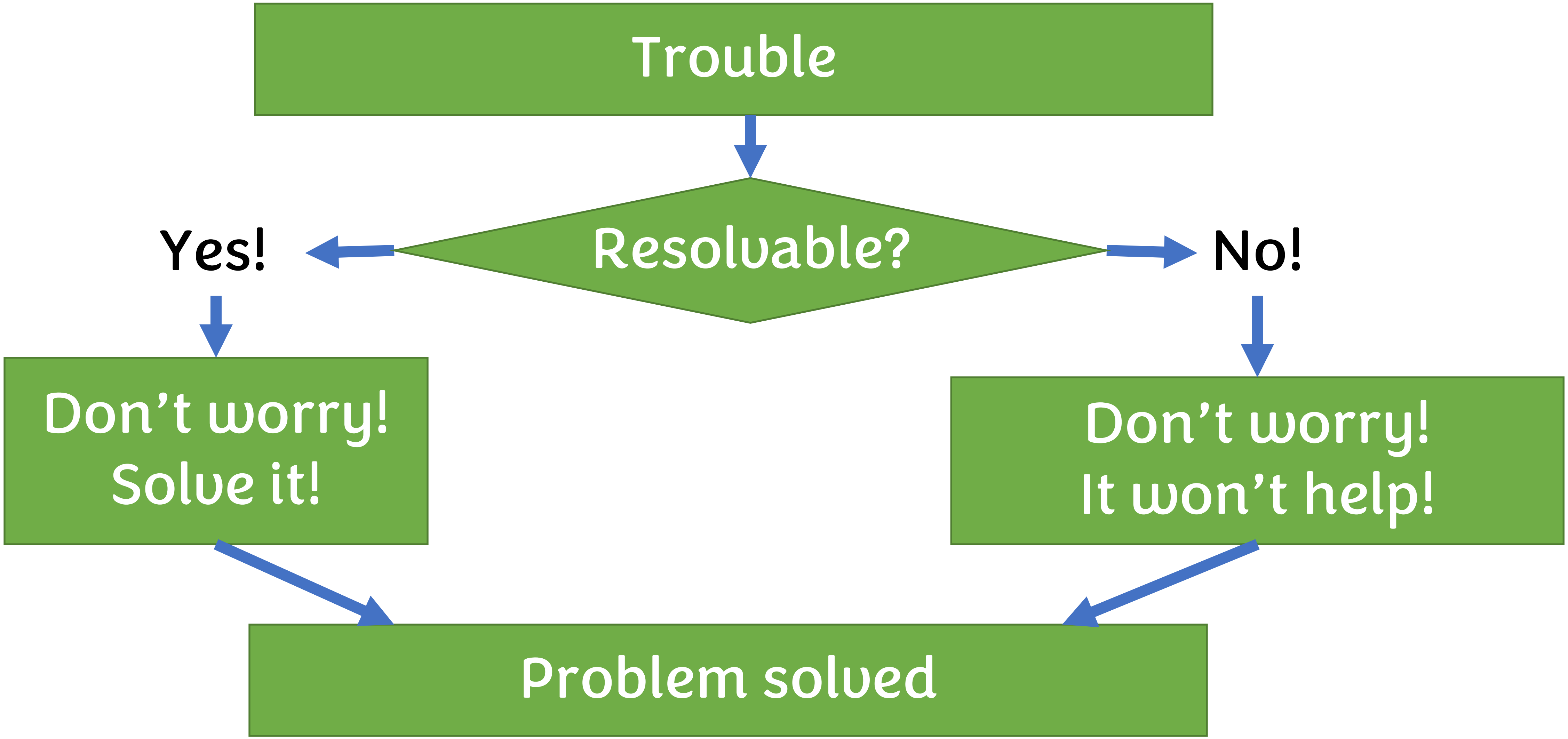
Collaboration: the “whiteboard” policy

- You are welcome to chat with each other (also welcome to come to OHs), but you come with nothing and leave with nothing
- When you type your answers / code, it must be done on your own. It must be close-book. It must be typed by you, word by word.
- Any violation may result in severe outcome. Usually -100% current/all homework assignment score, fail the course, report to the university, the university may make further decisions
- Must cite if any idea is from other sources, including people, books, websites, AI, etc.

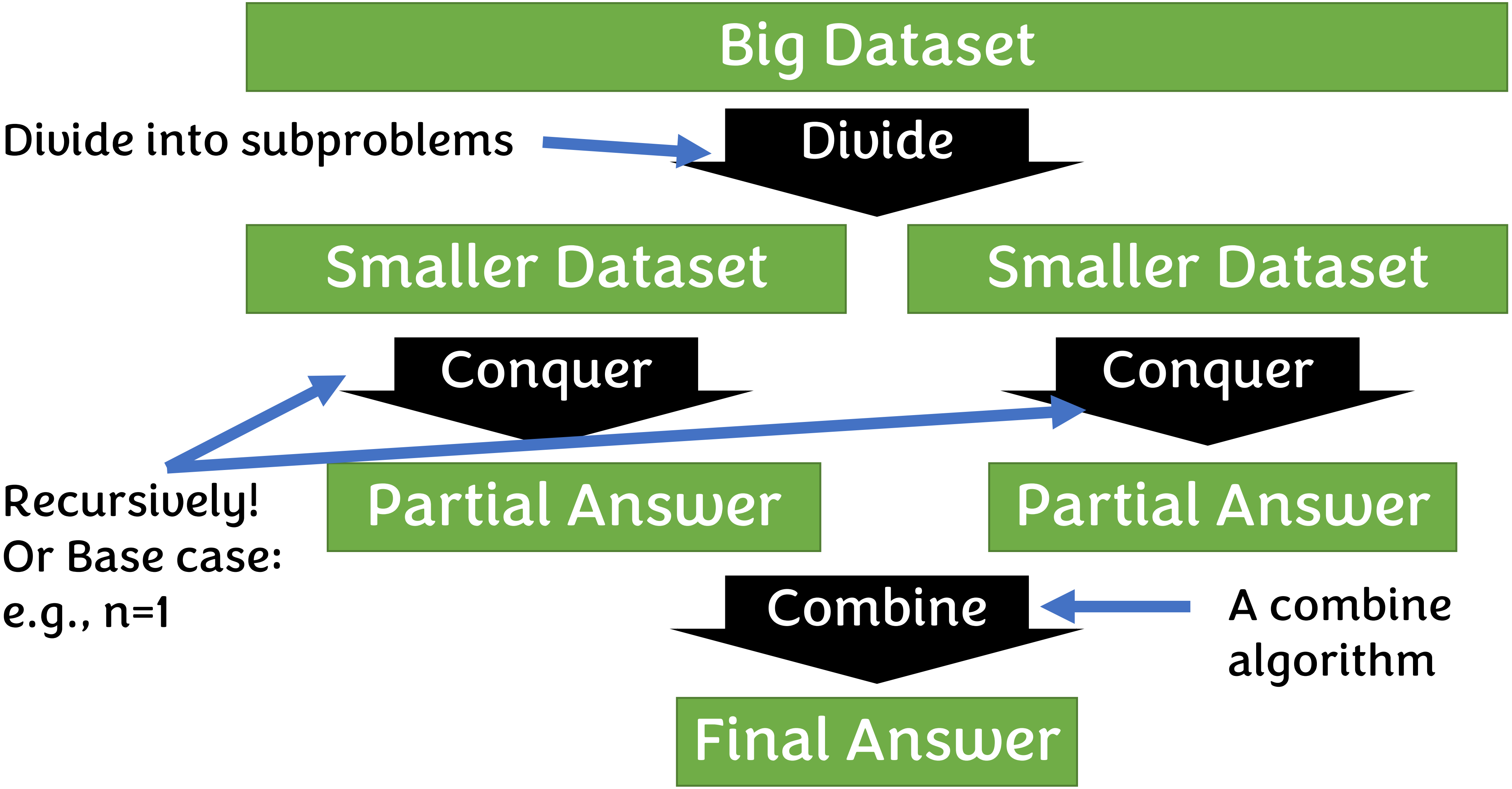
Analyzing algorithms & Divide-and-conquer

Yan Gu

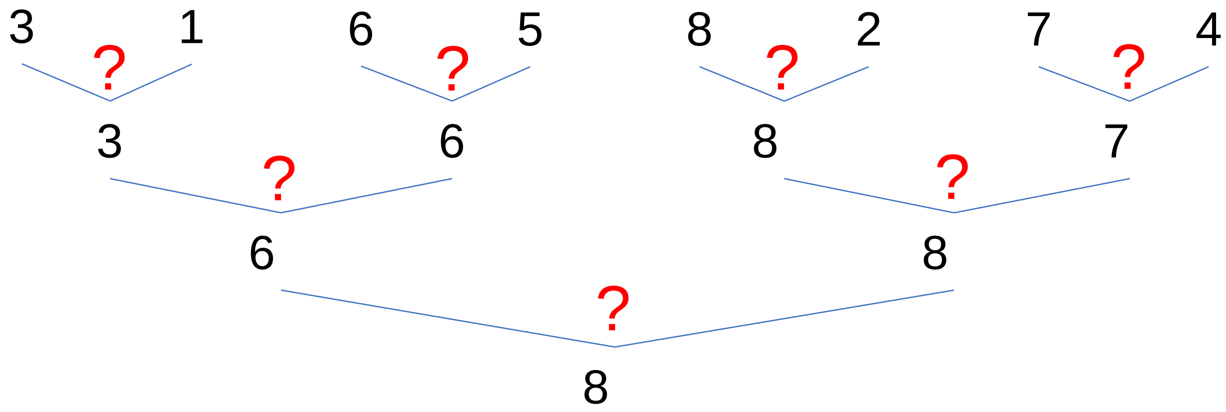
Divide-and-conquer in real-world



Classic Divide-and-Conquer



Find the max in an array using a tournament




```
int get_max (A[1..n]) {
  if (n==1) return A[1];
  m = n/2;
  x = get_max(A[1..m]);
  y = get_max(A[m+1..n]);
  return max(x, y);
}
```

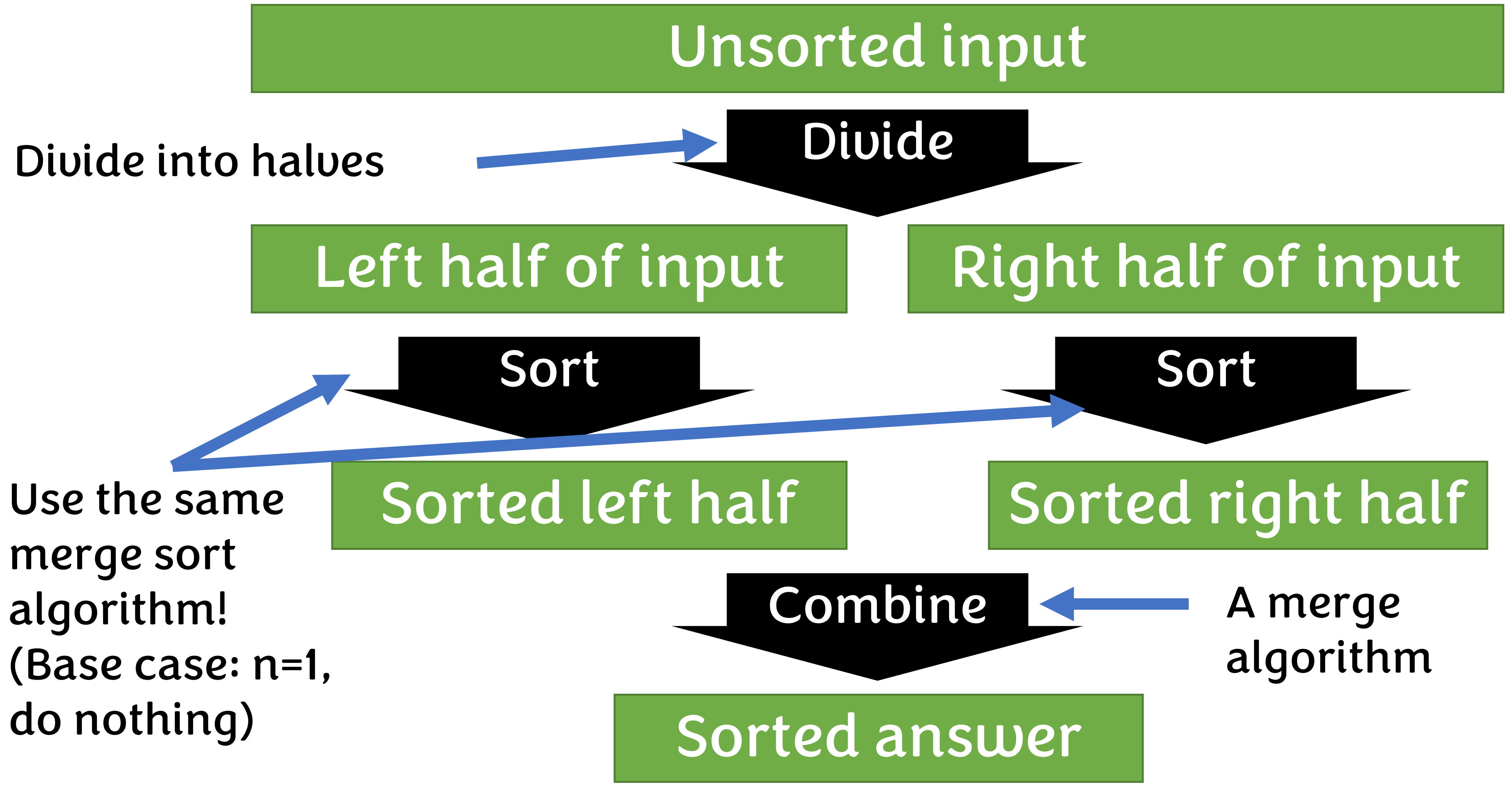
Base case

Divide-and-conquer

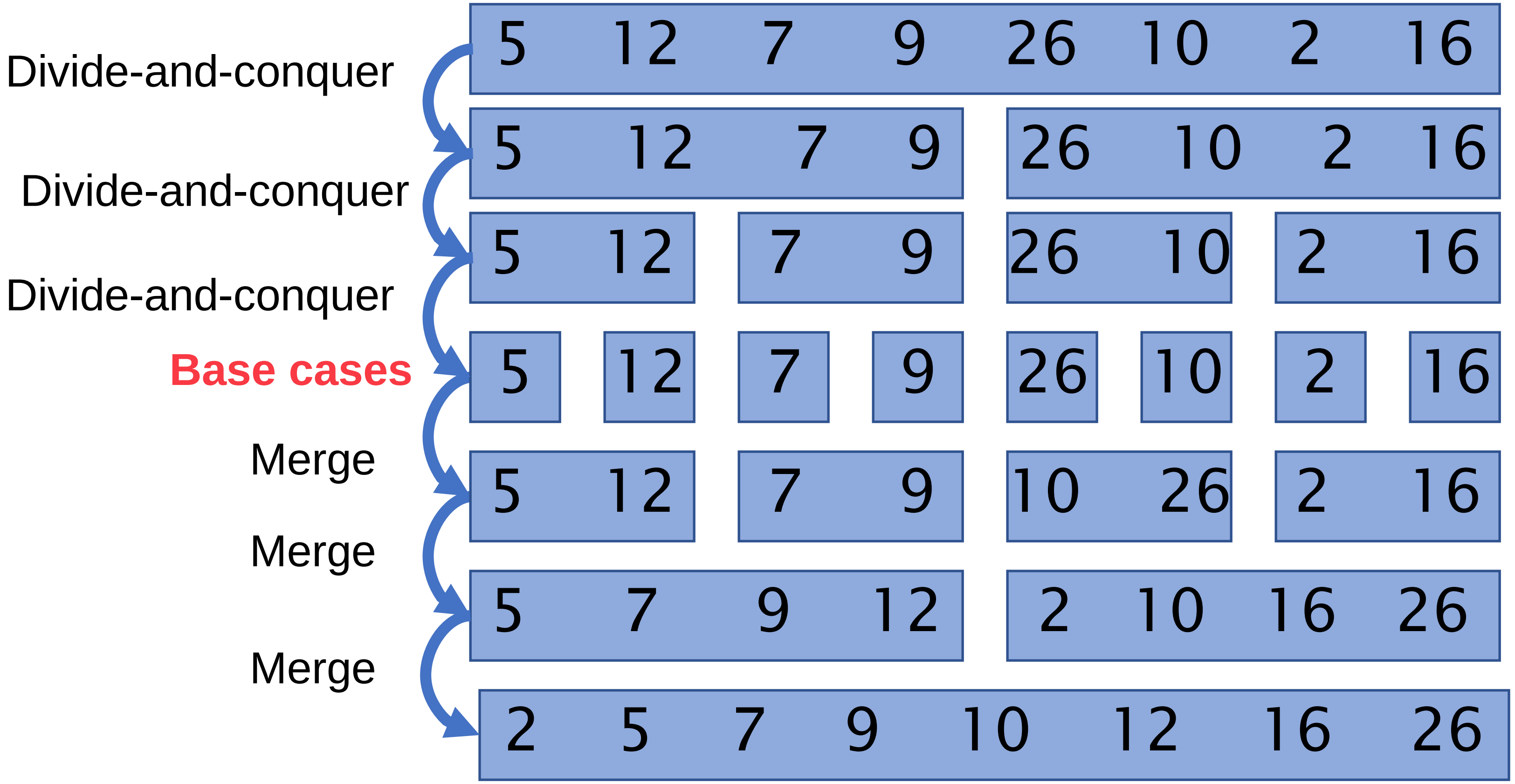
Combine

Merge sort

Divide-and-Conquer: merge sort



Merge sort




```
void mergesort(int *A, int n) {
  if (n <= 1) return; else {
    mergesort(A, n/2);
    mergesort(A+n/2, n-n/2);
    A = merge(A, n/2, A+n/2, n-n/2);  }}
```

← Divide

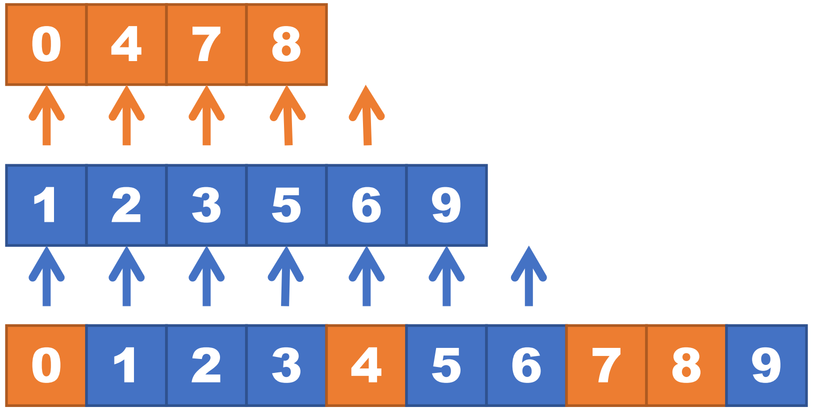
← Conquer

← Combine

Merge two sorted arrays

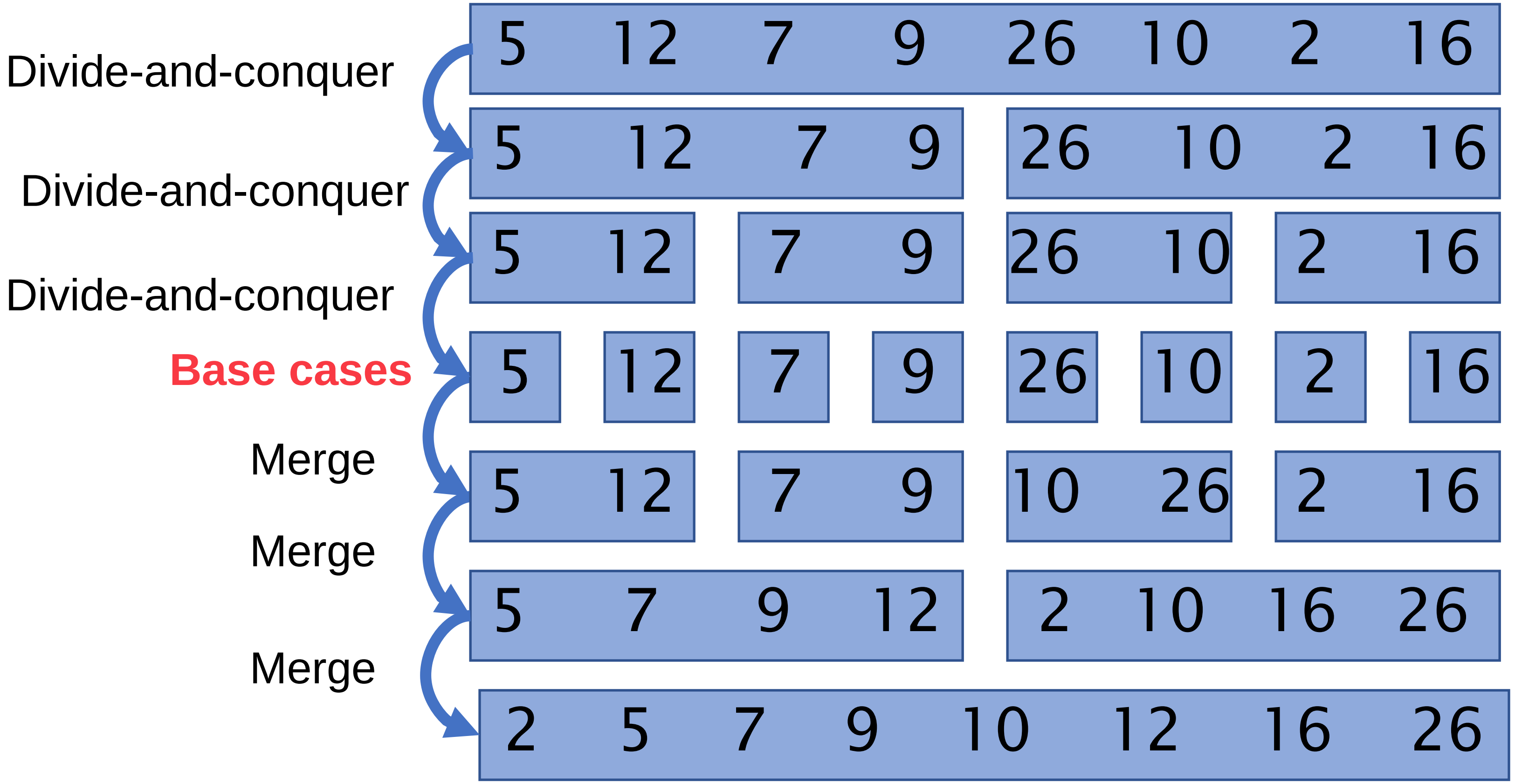
- Given two sorted arrays
- Combine them into one sorted one

```
merge(A, na, B, nb) {
  p1 = 0; p2 = 0; p3 = 0;
  while ((p1 < na) && (p2< nb)) {
    if (A[p1]<B[p2]) {
      C[p3++] = A[p1]; p1++;
    } else {
      C[p3++] = B[p2]; p2++;
    } }
  //copy the rest of the unfinished array
  return C;
}
```



- Costs $\Theta(n)$ time to merge two arrays of total size n

Merge sort




```
void mergesort(int *A, int n) {
  if (n <= 1) return; else {          ← Divide
    mergesort(A, n/2);
    mergesort(A+n/2, n-n/2);         ← Conquer
    A = merge(A, n/2, A+n/2, n-n/2);  }}  ← Combine
```

Merge sort Time complexity?

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + O(n) & \text{otherwise} \end{cases}$$

```
void mergesort(int *A, int n) {
  if (n <= 1) return; else {
    mergesort(A, n/2);
    mergesort(A+n/2, n-n/2);
    A = merge(A, n/2, A+n/2, n-n/2);  }}
```

Solve the recurrence

$$T(n) = \begin{cases} c & \text{if } n \leq 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$

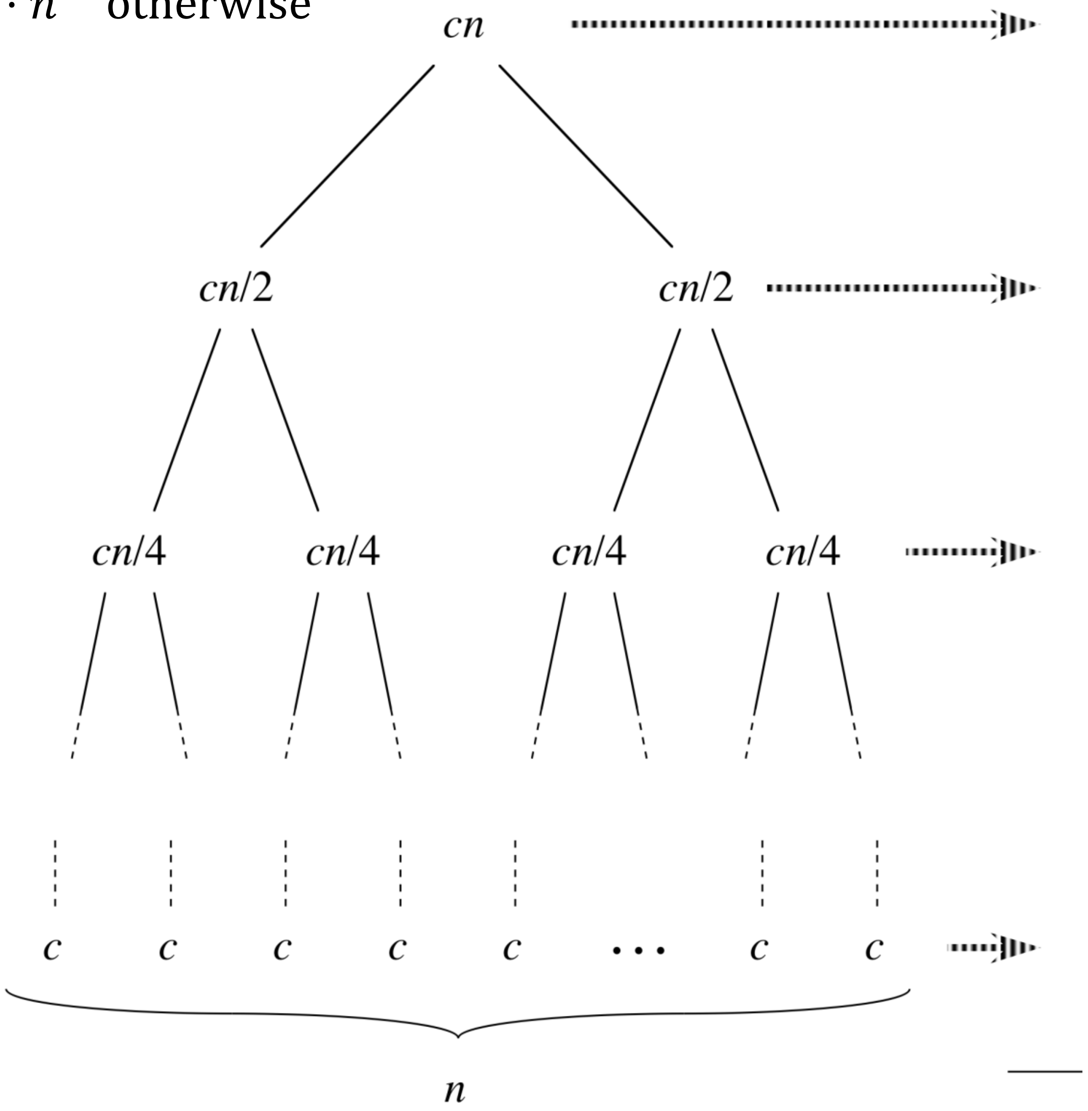
- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$
- $T(n) = 4T\left(\frac{n}{4}\right) + 2 \cdot \frac{n}{2} + n$
- $T(n) = 4\left(T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + n + n$
- …… (after $\log n$) levels
- $T(n) = 2^{\log n}\left(T(1) + \frac{n}{2^{\log n}}\right) + \log n \cdot n$
- $T(n) = n \log n$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$

Recursion Tree

$$T(n) = \begin{cases} c & \text{if } n \le 1, \\ 2T(n/2) + c \cdot n & \text{otherwise} \end{cases}$$



i	# nodes	Total cost
0	$1=2^0$	cn
1	$2=2^1$	cn
2	$4=2^2$	cn
…	…	…
d	$n = 2^d$ $d \approx \log n$	cn

How to solve a recurrence in general?

Solving recurrences – Master Theorem

- The Master Method for solving divide-and-conquer recurrences applies to the recurrences in the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where $a \geq 1, b > 1$, and f is asymptotically positive (positive for sufficiently large n).

Base case: $T(c)$ is a constant when c is a constant

Master Theorem

- Solve $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1$ and $b > 1$, f is asymptotically positive
- Let $y = \log_b a$ and constant $k \geq 0$. The leaf cost is $\Theta(n^y)$. The root cost is $f(n)$

- Case 1: $f(n) = O(n^{y'})$ for $y' < y$

leaf cost $\gg$ root cost $f(n)$ $\Rightarrow$ Leaf dominated (differ by at least n^ϵ)

$\Rightarrow T(n) = \Theta(n^y) =$ leaf cost

- Case 2: $f(n) = \Theta(n^y \log^k n)$

leaf cost $\approx$root cost $f(n)$ (can differ by at most a factor of $\log^k n$)

$\Rightarrow T(n) = \Theta\left(n^y \log^{k+1} n\right) = \Theta(f(n) \log n) =$ #levels $\times$ root cost

- Case 3: $f(n) = \Omega(n^{y'})$ for $y' > y$ and *regularity condition*

leaf cost $\ll f(n) \Rightarrow$ Root dominated (differ by at least n^ϵ)

$\Rightarrow T(n) = \Theta\left(f(n)\right) =$ root cost

If you are not familiar with this

- **Read CLRS Chapter 4: “Divide-and-Conquer”**
  - “The master method for solving recurrence”
  - “Proof of the master theorem”

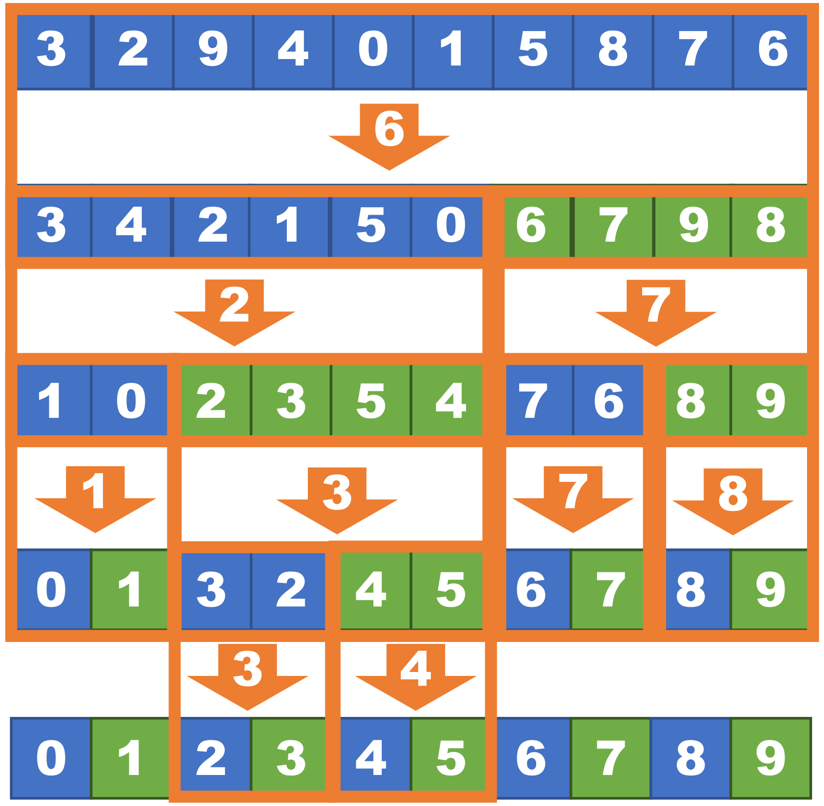
Quicksort

Quicksort

- **Another sorting algorithm that uses divide and conquer**
- **Divide (different from directly dividing into halves):**
  - Find a pivot x
  - Put all elements $\leq x$ on the left, call them L
  - Put all elements $\geq x$ on the right, call them R
  - (Those $= x$ can be put either on the right or left, or in the middle)
- **Conquer**
  - Sort $\boldsymbol{L}$ and $\boldsymbol{R}$ recursively
- **Combine**
  - No need to do anything

Quicksort

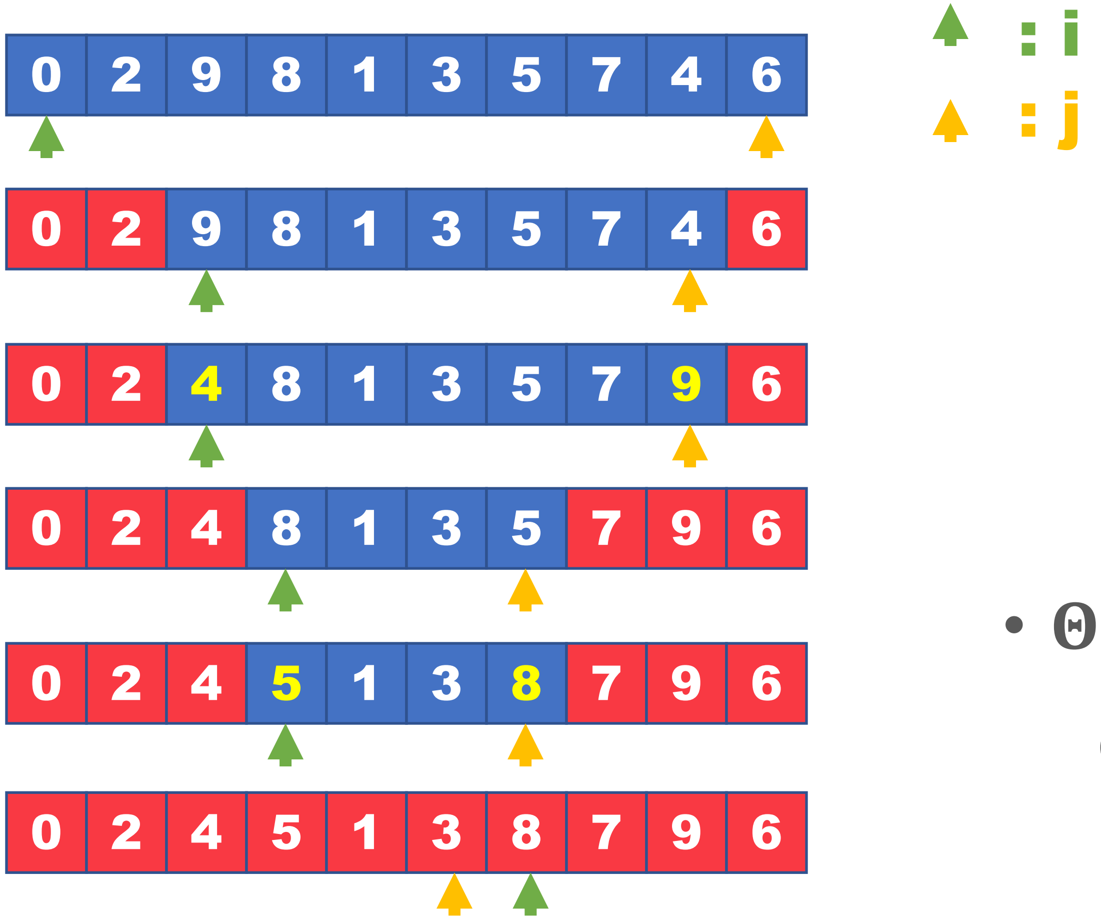
- Find a random pivot x in the array
- Put all elements in A that are $\leq p$ on the left of p, and all elements in A that are $\geq p$ on the right



- The hardest part is in how to partition!

Divide: Partition the array

- **How to move elements around? (using 6 as a pivot)**

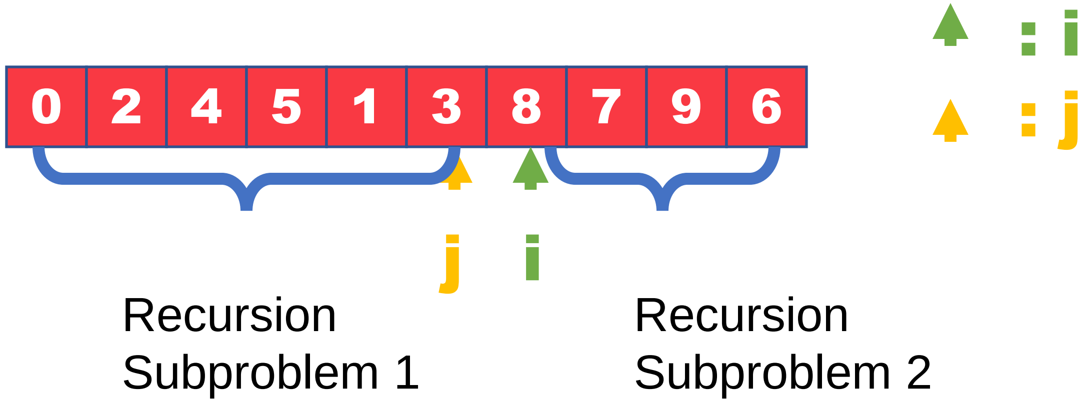



```
Partition(A, n, x) {
  i = 0; j = n-1;
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j++;
    if (i < j) {
      swap A[i] and A[j];
      i++; j--;
    }
  }
}
```

- **$\Theta(n)$ time for one round**
  (the pointers never move back, so each element is accessed at most once)

Conquer

- **How to move elements around? (using 6 as a pivot)**




```
qsort(A, n) {
  i = 0; j = n-1; x = A[rand(0,n)];
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j++;
    if (i < j) {
      swap A[i] and A[j];
      i++; j--;
    }
  }
  if (i < r-1) qsort(A+i, r);
  if (0 < j) qsort (A, j+1);
}
```

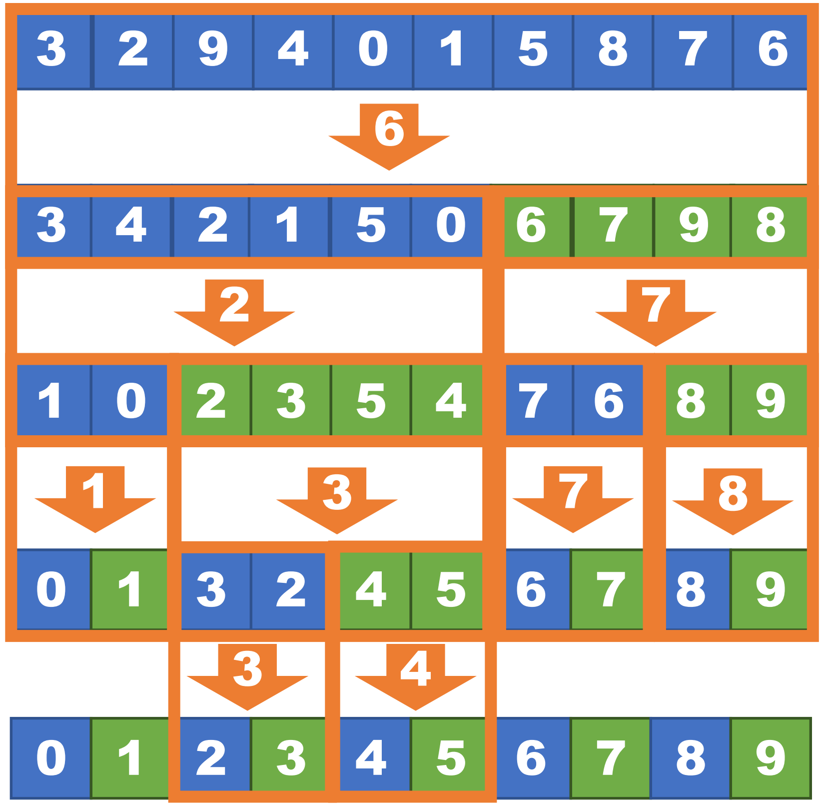
Divide (partition)

Conquer (recurse)

Quicksort – cost analysis

- **If every time we can partition the array perfectly in halves**
  - $O(\log n)$ rounds, $O(n \log n)$ time in total
- **But in the worst case, it is $O(n^2)$**
  - What is the worst case?

- **Does that mean it has similar performance as bubble sort/selection sort/insertion sort?**
- **The average cost is $O(n \log n)$!**
- **(we'll see more details later in the course)**



Matrix Multiplication

Matrix Multiplication

Consider standard iterative matrix-multiplication algorithm

$$C := A \; B \qquad\qquad C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

- Where A, B, and C are $N \times N$ matrices

```
for i = 1 to N do
    for j = 1 to N do
        for k = 1 to N do
            C[i][j] += A[i][k] * B[k][j]
```

- $\Theta(N^3)$ computation in RAM model.

Recursive Matrix Multiplication

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} := \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Compute 8 submatrix products recursively

$C_{11} := A_{11}B_{11} + A_{12}B_{21}$

$C_{12} := A_{11}B_{12} + A_{12}B_{22}$

$C_{21} := A_{21}B_{11} + A_{22}B_{21}$

$C_{22} := A_{21}B_{12} + A_{22}B_{22}$

- **8-way divide-and-conquer**
  - $T(N) = \Theta\left(N^2\right) + 8T\left(\frac{N}{2}\right)$
  - **=> solution:** $T(N) = \Theta(N^3)$
  - **[No improvement in theory, is it useful?]**

Strassen's Algorithm: cost analysis

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} := \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Step 1: Compute matrices S:

- $S_1 = B_{12} - B_{22}$ $\quad S_6 = B_{11} + B_{22}$
- $S_2 =$ ... $_{22}$
- $S_3 =$ ... $_{22}$
- $S_4 = B_{21} - B_{11}$ $\quad S_9 = A_{11} - A_{21}$
- $S_5 = A_{11} + A_{22}$ $\quad S_{10} = B_{11} + B_{12}$

Simple + and - matrices

Size $N/2$

Total cost $\Theta(N^2)$

Step 3: Compute the matrix C:

- $C_{11} = P_5 + P_4 - P_2 + P6$
- $C_{22} = P_5 + P_1 - P_3 - P_7$

Simple + and - matrices

Size $N/2$

Total cost $\Theta(N^2)$

Step 2: Compute P matrices:

Matrix $\times$, calculated recursively

Cost: $7T(N/2)$

- $P_5 = S_5 \cdot S_6$ $\quad P_6 = S_7 \cdot S_8$
- $P_7 = S_9 \cdot S_{10}$

Only 7 of them!!

Let $T(N)$ be the cost of matrix multiplication on two matrices of $N \times N$

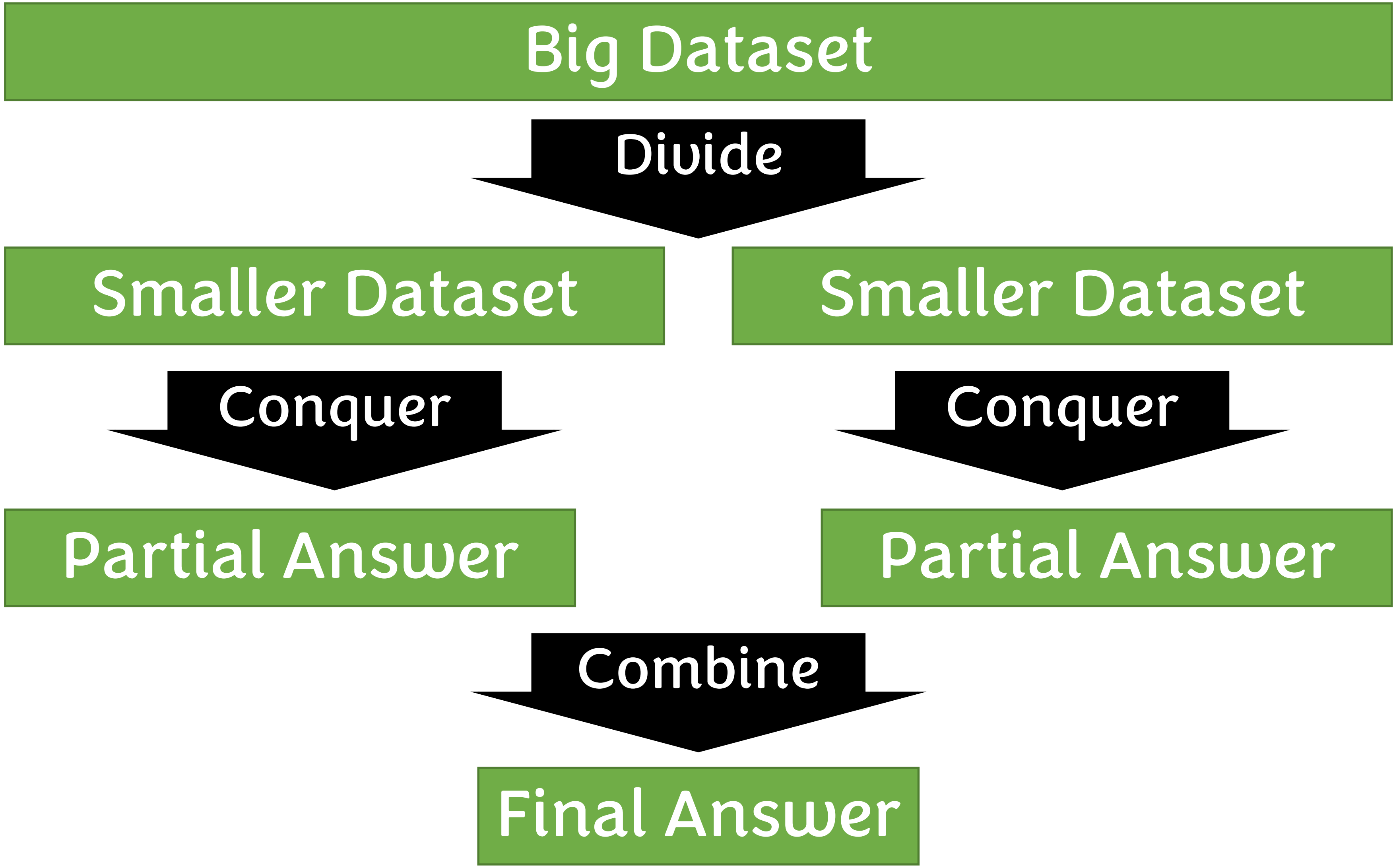
Strassen's Algorithm: cost analysis

- $T(N) = 7T\left(\frac{N}{2}\right) + cN^2$

- **Solve it using Master Theorem**

- **Solution:**
- $T(N) = \Theta\left(N^{\log_2 7}\right) \approx \Theta(N^{2.8074})$
  - Smaller than N^3!
  - Computing the multiplication of two matrices of size N doesn't need $\Theta(N^3)$ operations!
  - The best-known algorithm today is $\Theta(N^{2.3728596})$

Classic Divide-and-Conquer



There are many other ideas that generally can be viewed as "divide-and-conquer"

Decrease-and-conquer

Decrease-and-conquer

- **Divide the problem into subproblems**
- **Narrow down the solution in one or several subproblems**
  - We do not need to go into all subproblems!
- **Reduce time complexity by smartly partitioning the problem!**

- **Example: binary search**
  - Given a sorted array and an element x, decide if x is in the array / its rank in the array
  - Naïvely: compare from the first element, $O(n)$ time complexity
  - Binary search: $O(\log n)$ comparisons at most

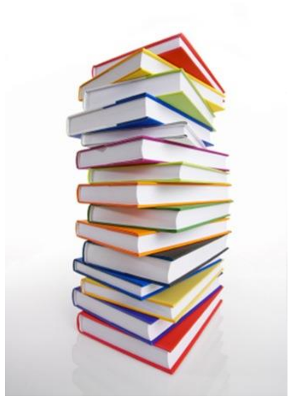
2	5	7	8	10	15	22	23	29	40

$x = 29$

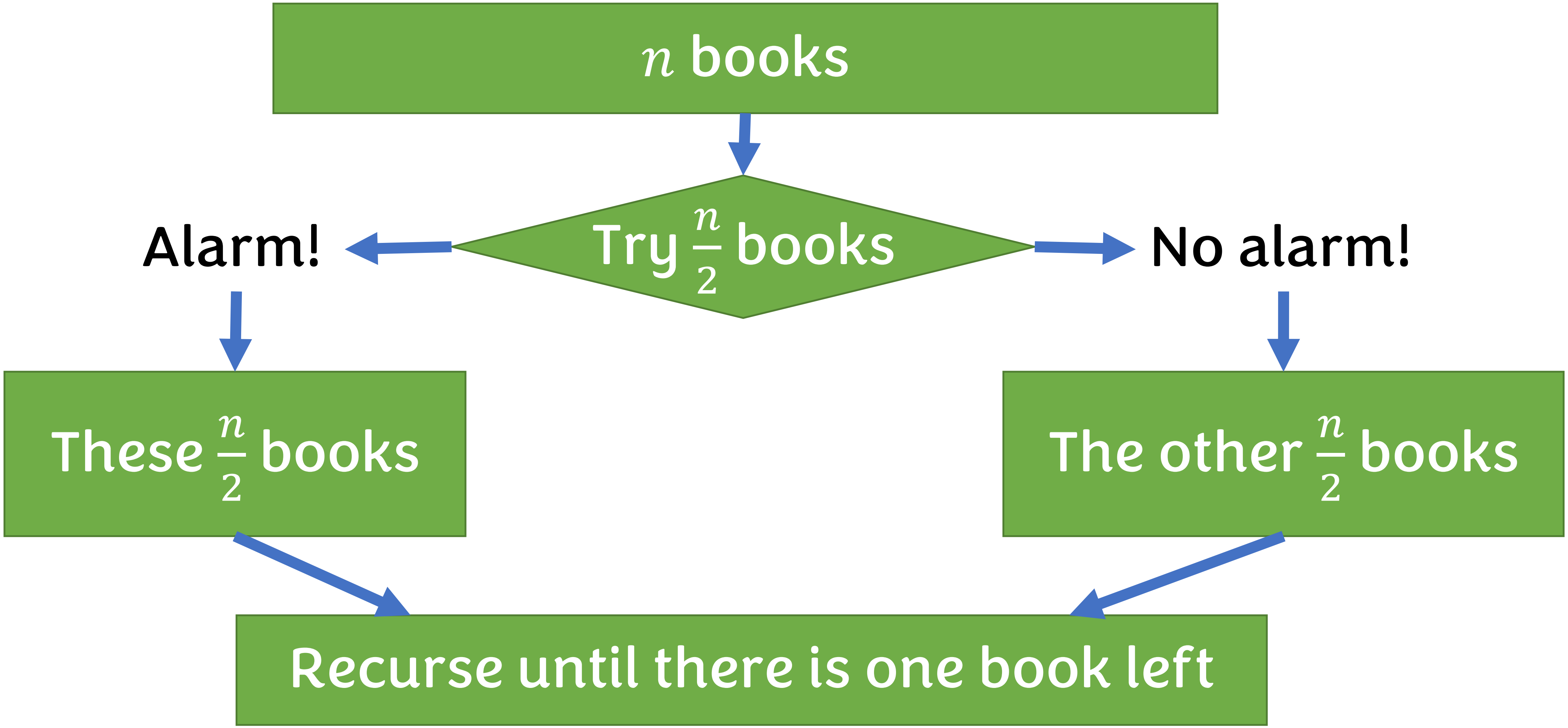
We use divide-and-conquer to solve real-world problems

- **You went to the library, and borrowed quite a few books. Unfortunately, one of the books was not checked out, so the alarm started.**
  - You need to find the book and check it out again
- **You can try each book and see if this book was not checked out.**

- **Any better solution?**



Decrease-and-conquer



Another example: COVID testing

- **How to run COVID test for all citizens in a city with a million population?**
- **Assumption: we know that there are only 100 positive cases. How can we test it efficiently?**
- **Solution:**
  - They mix the parts of every 100 samples and test them (round 1)
  - For all positive samples, they check each of the samples in those groups (round 2)
  - Assume the city has 1 million people, how many tests do they need in the worst case?
  - 10^6/100 (round 1) + 100*100 (round 2, worst case) = 20000, saved 98% tests

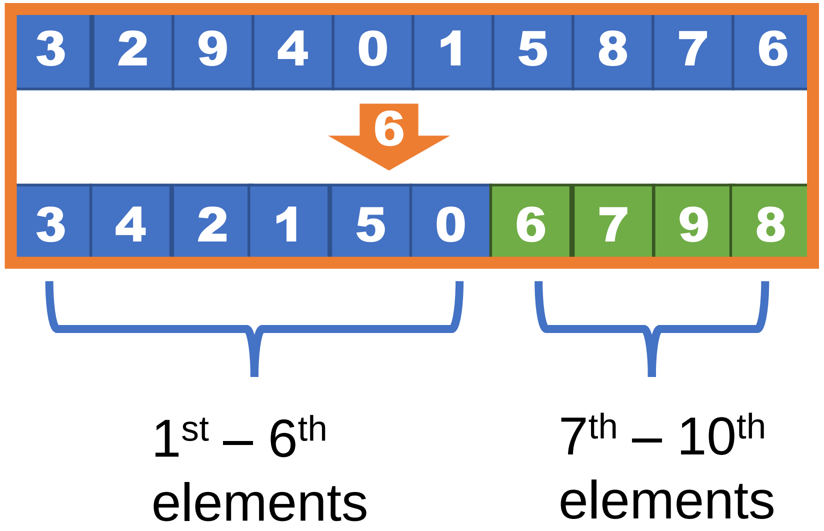
Quick selection

- **Given an unsorted array, how can I find the k-th element?**

- **Sort the array, output the $\boldsymbol{k}$-th element?**
  - $O(n \log n)$ time complexity
  - Somehow did some redundant work... We do not need to sort the array…

Quick selection

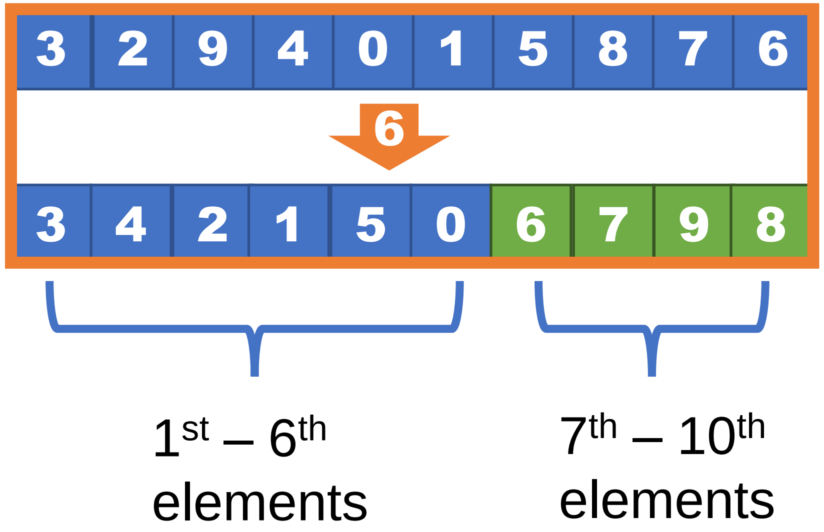
- Find a random pivot x in the array
- Put all elements in A that are $\leq p$ on the left of p, and all elements in A that are $\geq p$ on the right



- Whatever k is, it falls in exactly one subproblem!
- Let's only recurse on that part!
- Example:
  - If k=3: find the 3rd element in the first part
  - If k=8: find the (8-6=)2nd element in the second part
  - Recurse using the same algorithm!

Quick selection [CLRS 9.2]

- Whatever k is, it falls in exactly one subproblem!
- Let's only recurse on that part!



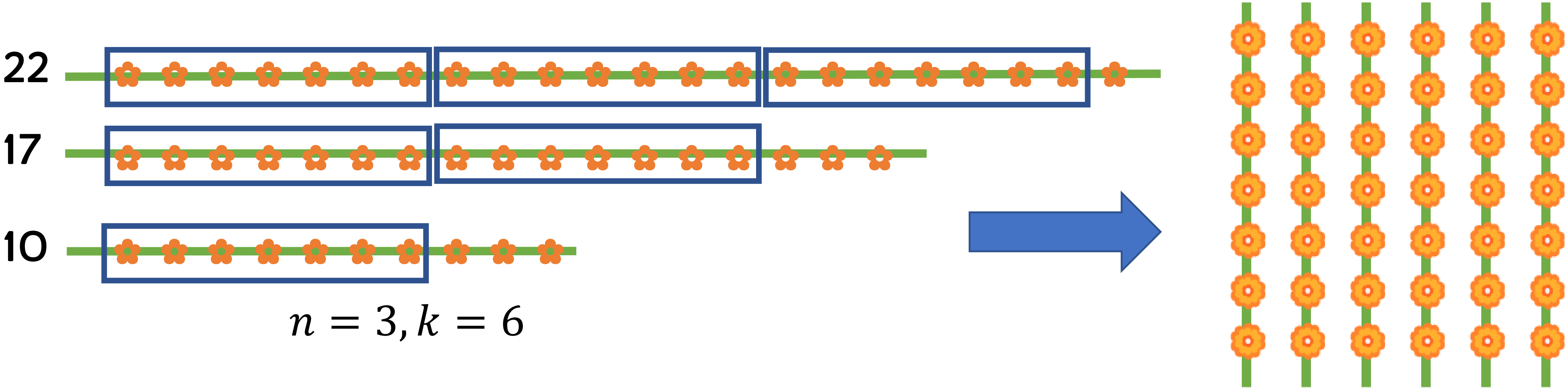

```
qsel(A, n, k) {
  if n=1 return A[0];
  i = 0; j = n-1; x = A[rand(0,n)];
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j++;
    if (i < j) {
      swap A[i] and A[j];
      i++; j--;
    }
  }

  if (i < k) return qsel(A+i, n-i, k-i);
         else return qsel(A, j+1, k);
}
```

Answer binary search

- Sometimes directly computing the optimal solution is extremely hard (optimization)
- However, checking whether a solution is valid is simple
- The solution set is “monotonic”

Flower Vines

- You want to build a flower vine to decorate your home!
- You have n strings of flowers, each of different length
- You plan to cut them into k **segments**, to let each segment with the same number of flowers
  - (each segment must be from the same string)
- You want the segment to be **as long as possible**!

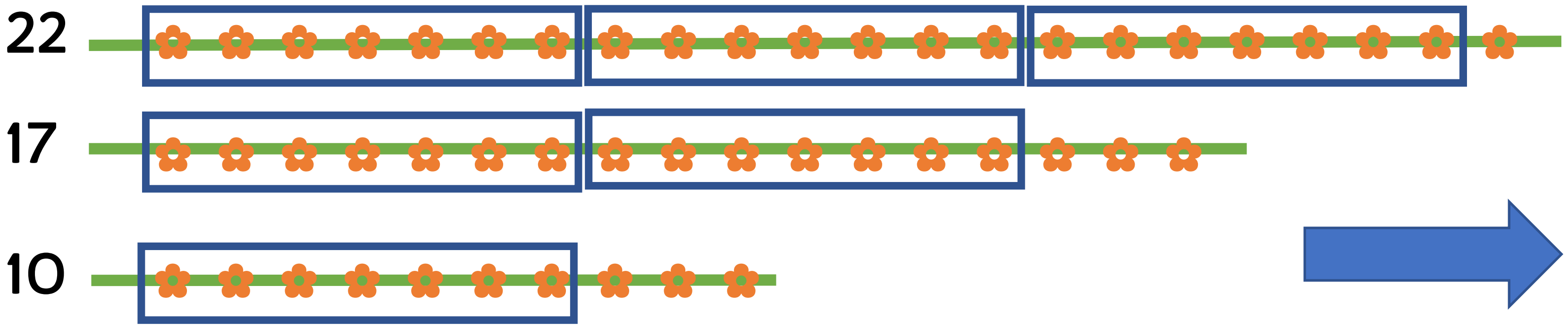


6 ×segments, each of 7

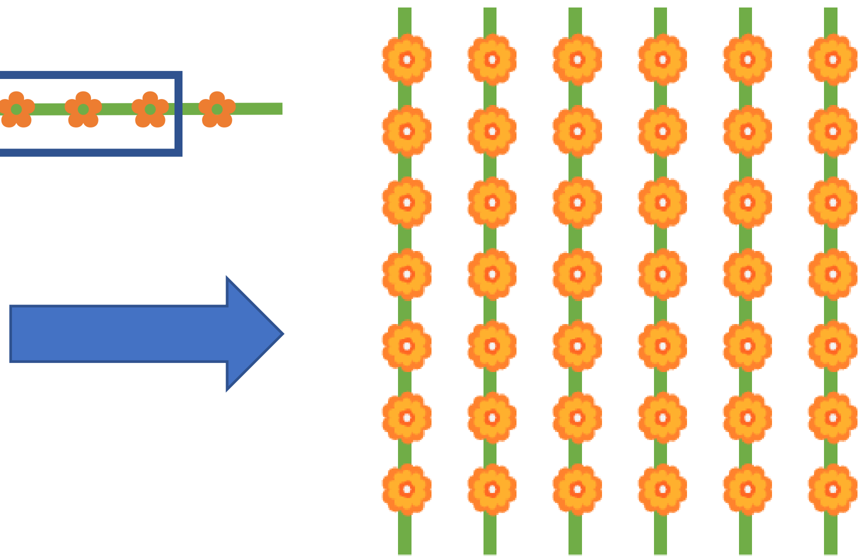
What is the longest length you can get?

Flower Vines

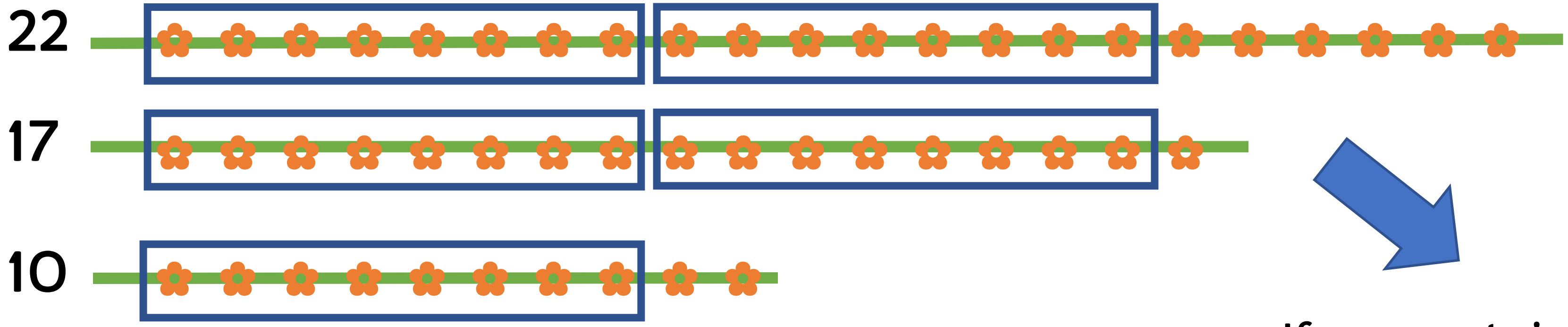
- Easy verification **(is x *feasible*?)** : given x, can we get k segments of size x?



$n = 3, k = 6$



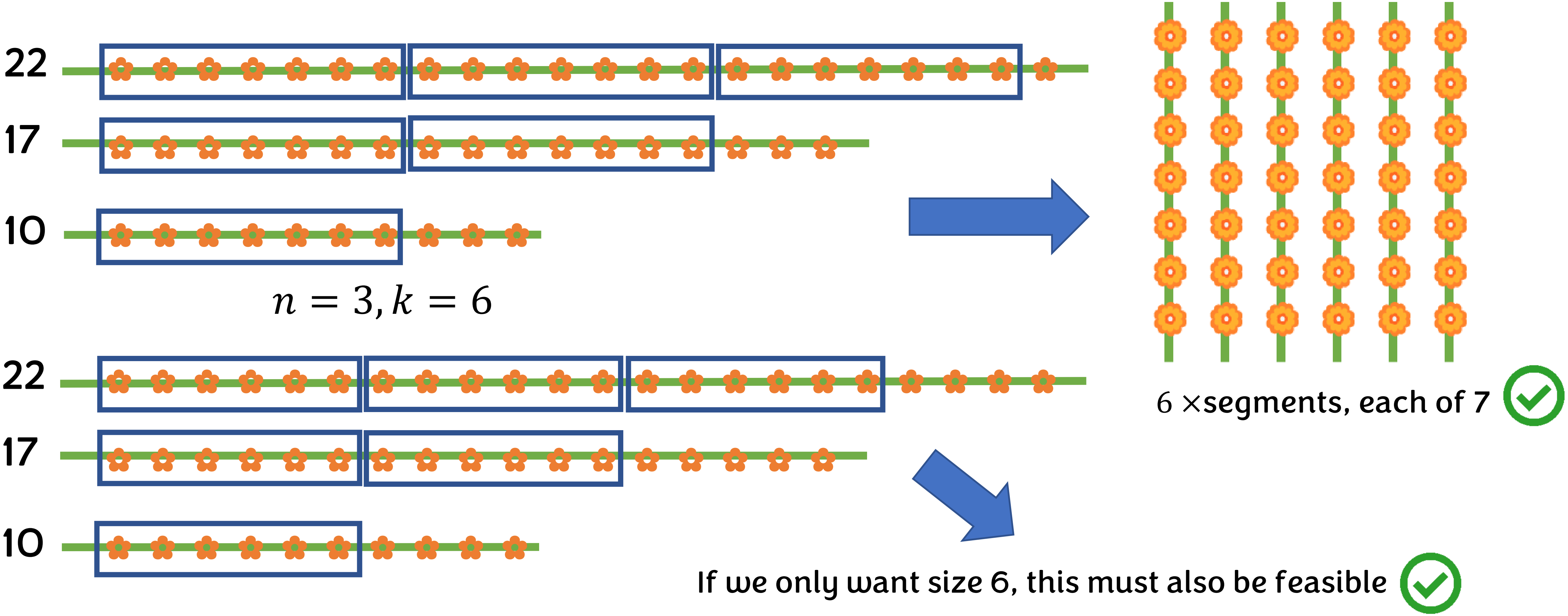
6 $\times$segments, each of 7



If we want size 8, we can only get 5

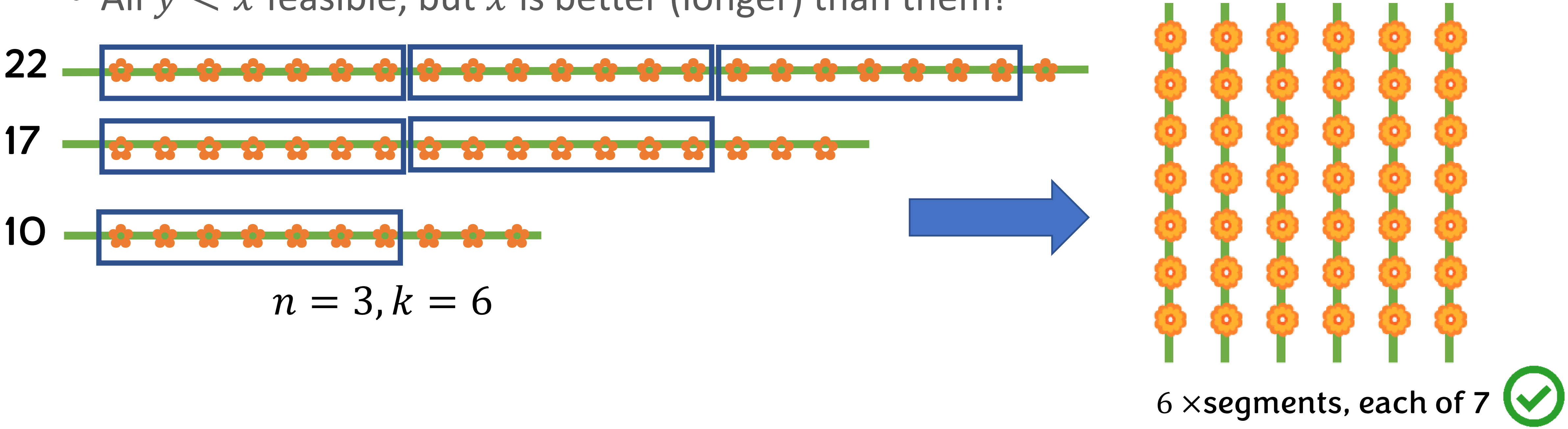
Flower Vines

- Monotonicity: if x is a solution, $y < x$ is also a solution



Flower Vines

- Easy verification: given x, can we get k segments of size x?
- Monotonicity: if x is a solution, $y < x$ is also a solution
- **Optimal** solution x means:
  - $x + 1$ not feasible
  - x feasible
  - All $y < x$ feasible, but x is better (longer) than them!



Flower Vines

- Easy verification: given x, can we get k segments of size x?
- Monotonicity: if x is a solution, $y < x$ is also a solution
- **Optimal** solution x means:
  - $x + 1$ not feasible
  - x feasible
  - All $y < x$ feasible, but x is better (longer) than them!

```
for (x = 0; x < max_possible; x++) {
  if (feasible(x)) continue; // x feasible! Try the next one!
  else break; // x not feasible!
}
return x-1; // x-1 is feasible but x is not, so x-1 is the answer!
```

This may be expensive – can we make it faster?

Flower Vines

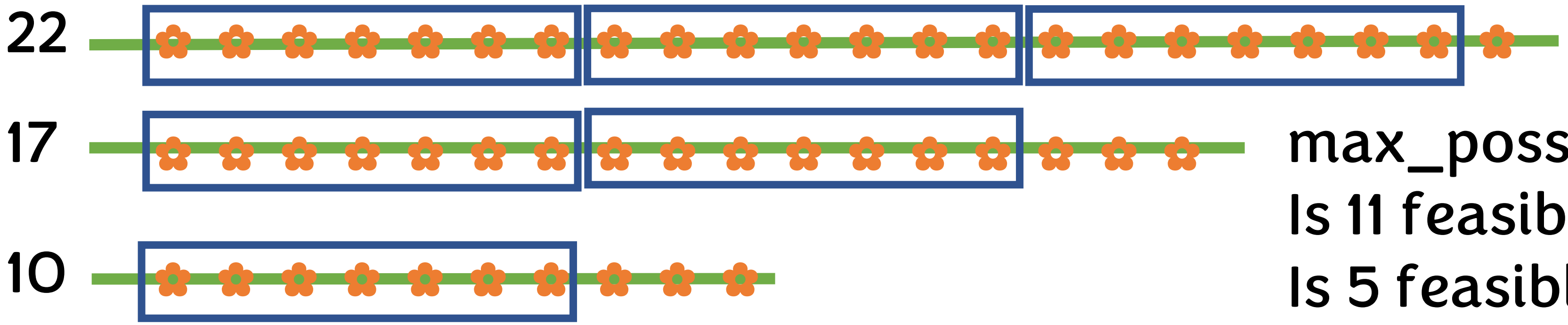
- Easy verification: given x, can we get k segments of size x?
- Monotonicity: if x is a solution, $y < x$ is also a solution
- **Optimal** solution x means:
  - $x + 1$ not feasible
  - x feasible
  - All $y < x$ feasible, but x is better (longer) than them!

```
int l = 0, r = max_possible+1, mid;
while (l < r-1) {
        mid = (l+r+1)/2;
        if (feasible(mid)) l = mid; // mid is feasible, best solution in [mid,r)
        else r = mid-1; // mid is not feasible, best solution in [l,mid-1]
}
return l;
```

Check the answer using binary search!

Flower Vines

```
int l = 0, r = max_possible+1, mid;
while (l < r-1) {
        mid = (l+r+1)/2;
        if (feasible(mid)) l = mid; // mid is feasible, best solution in [mid,r)
        else r = mid-1; // mid is not feasible, best solution in [l,mid-1]
}
return l;
```



$n = 3, k = 6$

max_possible = 22
Is 11 feasible? No => check 0 to 10
Is 5 feasible? Yes => check 5 to 10
Is 7 feasible? Yes => check 7 to 10
Is 8 feasible? No => check 7 to 8
$l = r - 1$ => l is the answer!

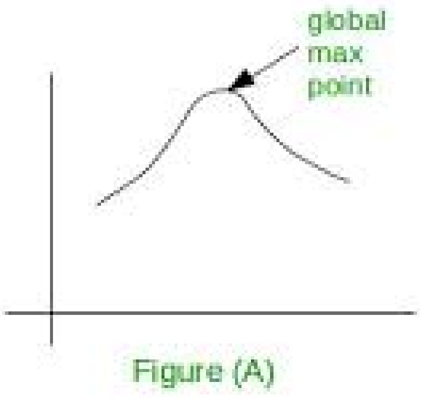
Binary search answers

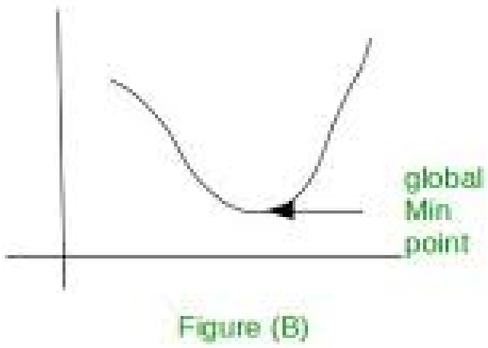
```
int l = 0, r = max_possible+1, mid;
while (l < r-1) {
        mid = (l+r+1)/2;
        if (feasible(mid)) l = mid; // mid is feasible, best solution in [mid,r)
        else r = mid-1; // mid is not feasible, best solution in [l,mid-1]
}
return l;
```

- **What if we do not know an upper bound (max_possible) of the solution?!**
- **Try in a sequence that increases exponentially: 1, 2, 4, 8, 16, …**
  - Increase the step as the possible answer gets larger
  - When we find a flip of feasible/infeasible, we now have a lower and upper bound – use regular binary search!
- **E.g., when answer is 29:**
  - 1(✓)→2(✓)→4(✓)→8(✓)→16(✓)→32(x)→24(✓)→28(✓)→30(x)→ 29(✓)
- Complexity: $O(\log ans)$, **ans is the answer**

Ternary Search

Unimodal functions





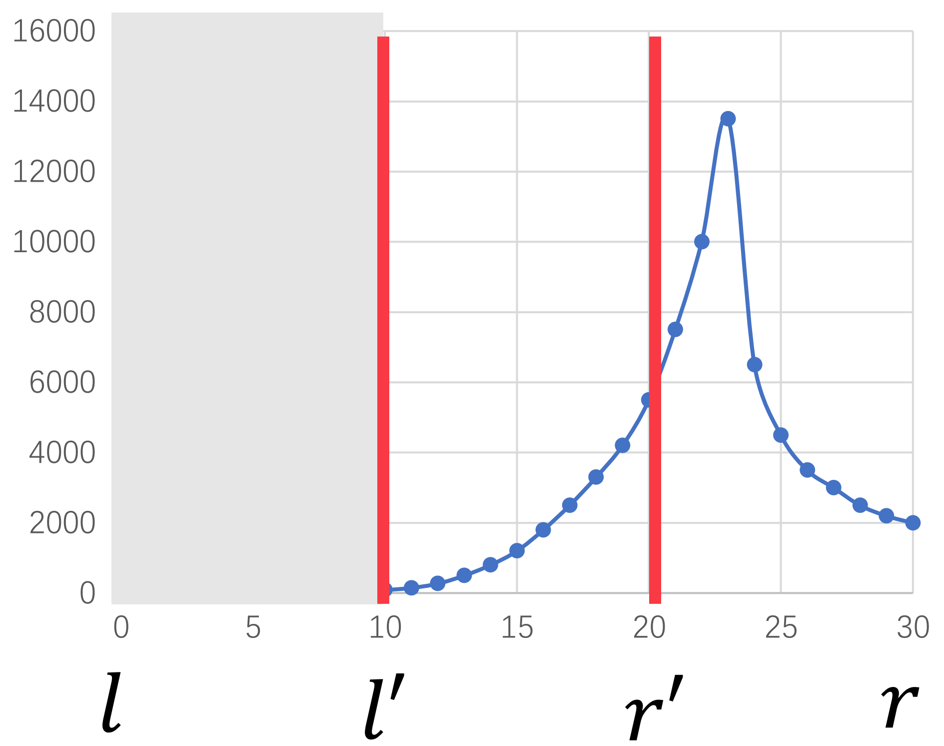
- Function with a single min/max point (unimodal function)
- Examples?

- Chemical reaction rate vs. temperature
- The taste of rice vs. the cooking time
- Running time of parallel algorithms vs. graunularity

- How to find where the min/max is?

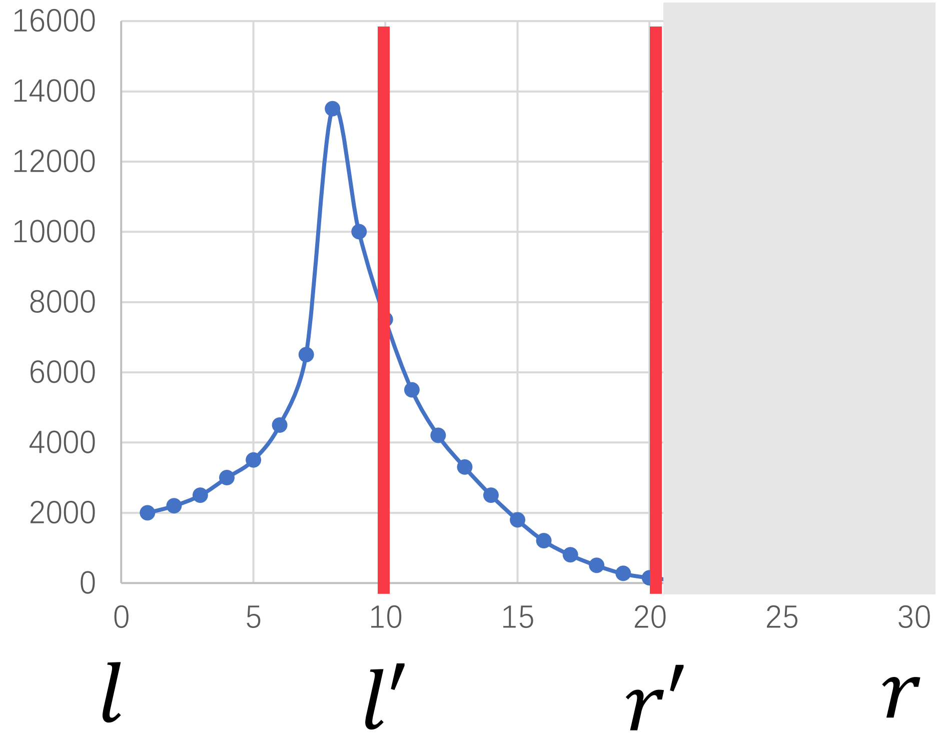
Ternary Search for max

- **When working on a range (l, r)**
- **Find the 1/3-th position and the 2/3-th**
- **Let $\boldsymbol{l' = l + (r - l)/3}$**
- **Let $\boldsymbol{r' = l + 2(r - l)/3}$**
- **If $\boldsymbol{f(l') < f(r')}$:**
  - narrow down the range to (l', r)
- **If $\boldsymbol{f(l') > f(r)}$:**
  - narrow down the range to (l, r')
- **(Drop the part on the smaller side)**
- **When l and r are sufficiently close, report**



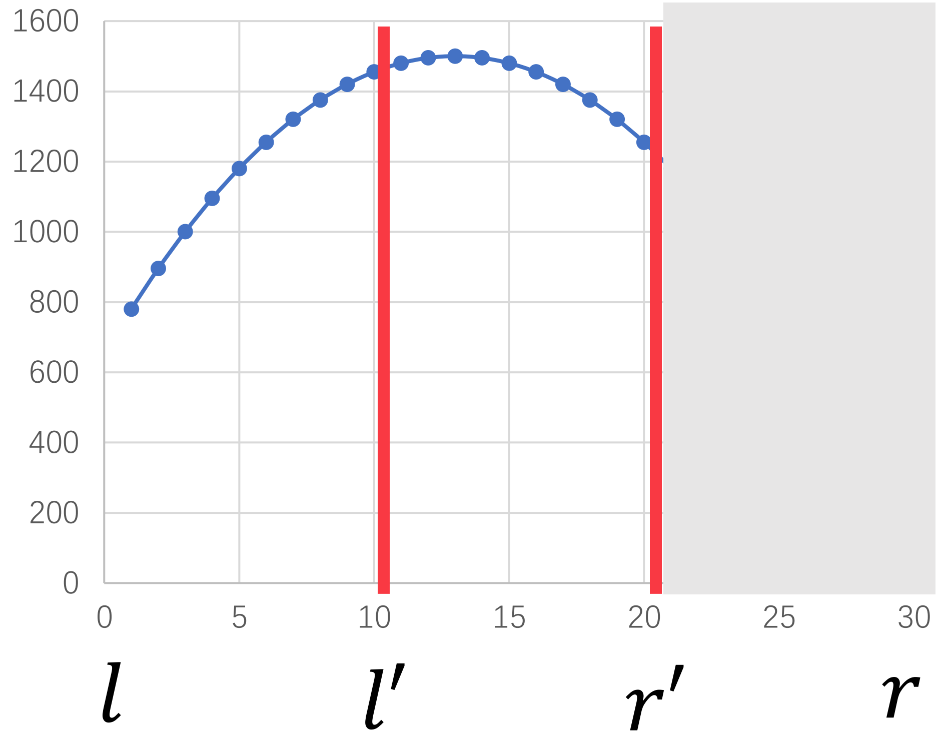
Ternary Search for max

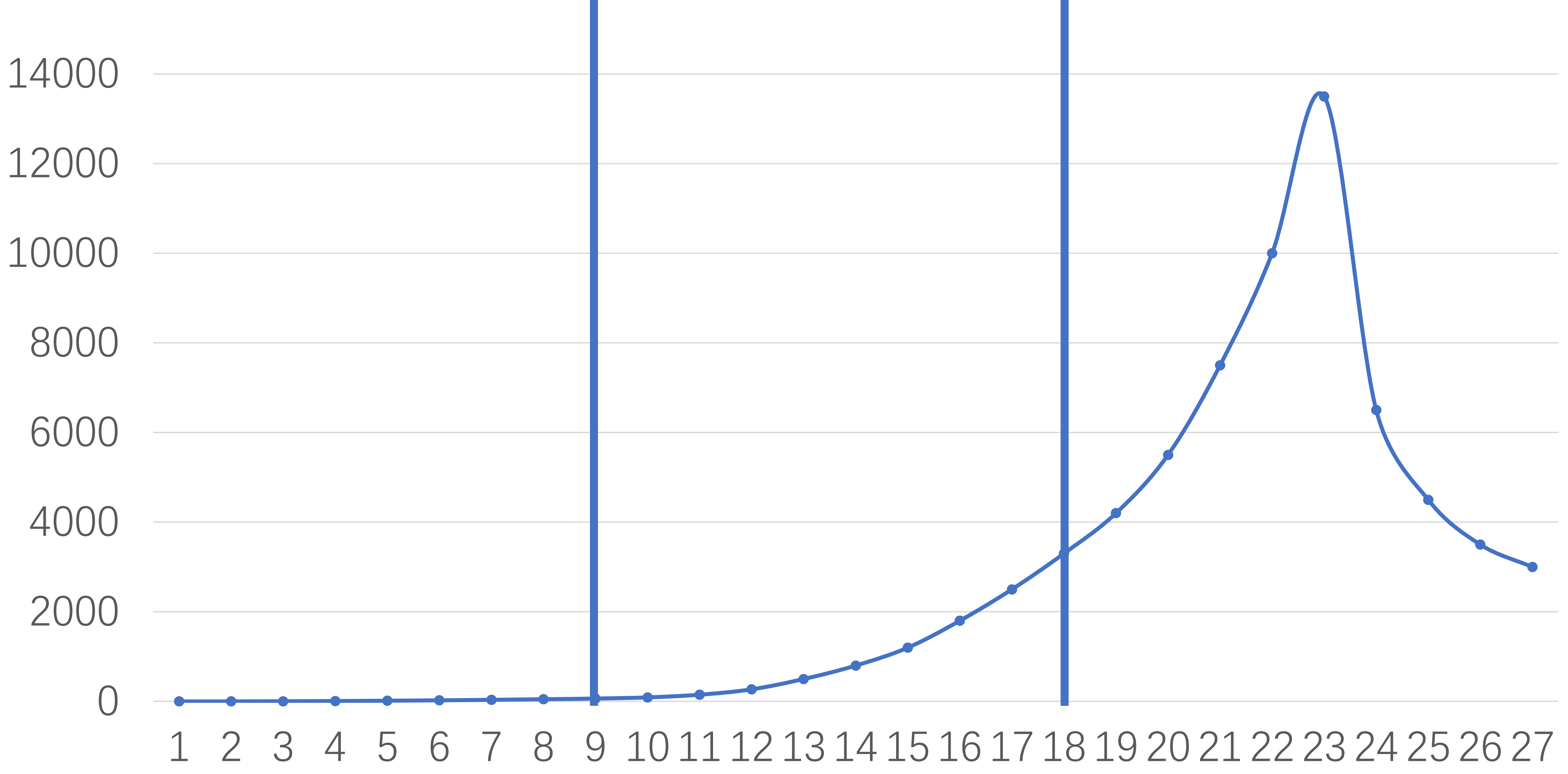
- **When working on a range (l, r)**
- **Find the 1/3-th position and the 2/3-th**
- **Let $l' = l + (r - l)/3$**
- **Let $r' = l + 2(r - l)/3$**
- **If $f(l') < f(r')$:**
  - narrow down the range to (l', r)
- **If $f(l') > f(r)$:**
  - narrow down the range to (l, r')
- **(Drop the part on the smaller side)**
- **When l and r are sufficiently close, report**

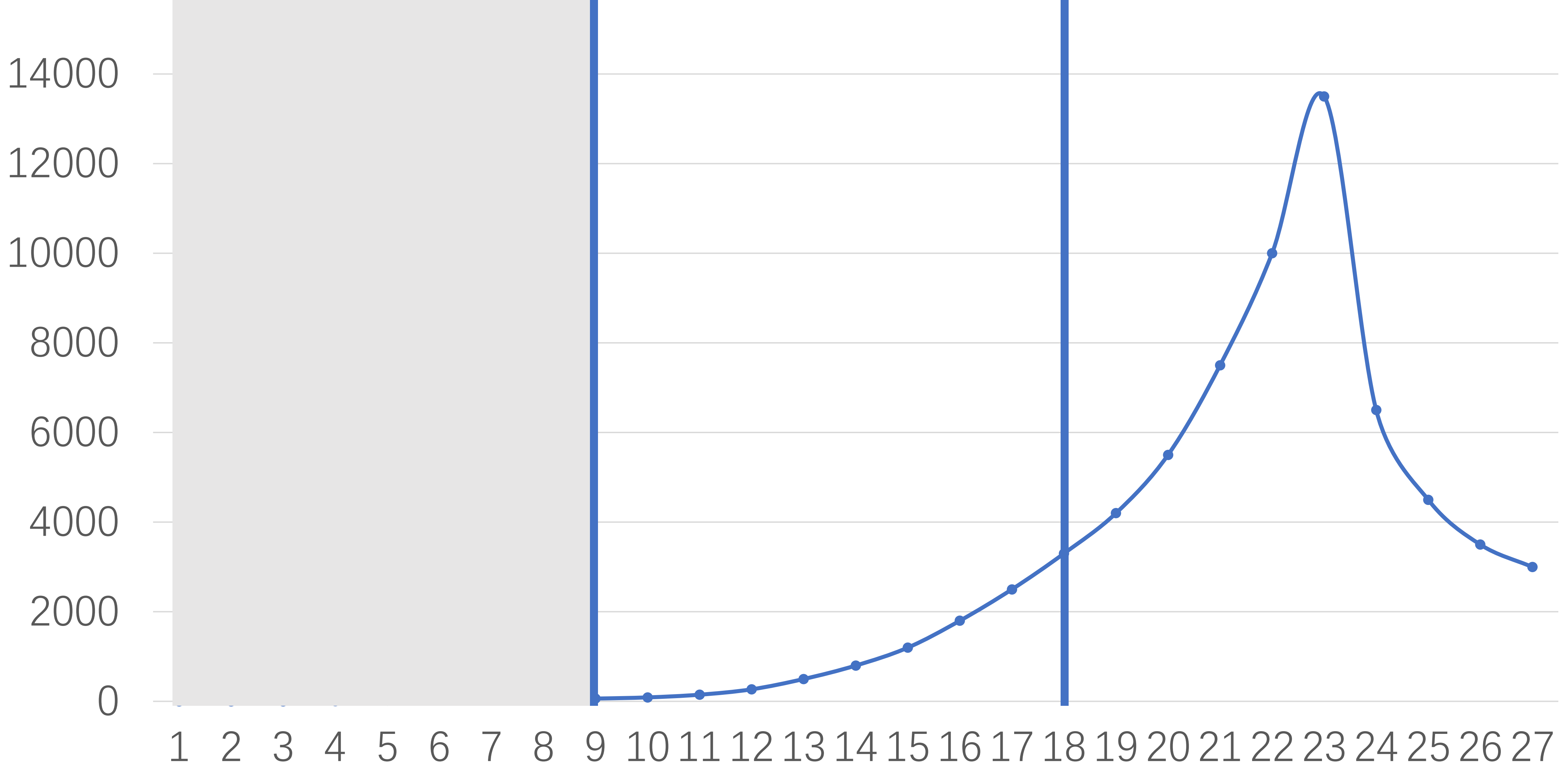


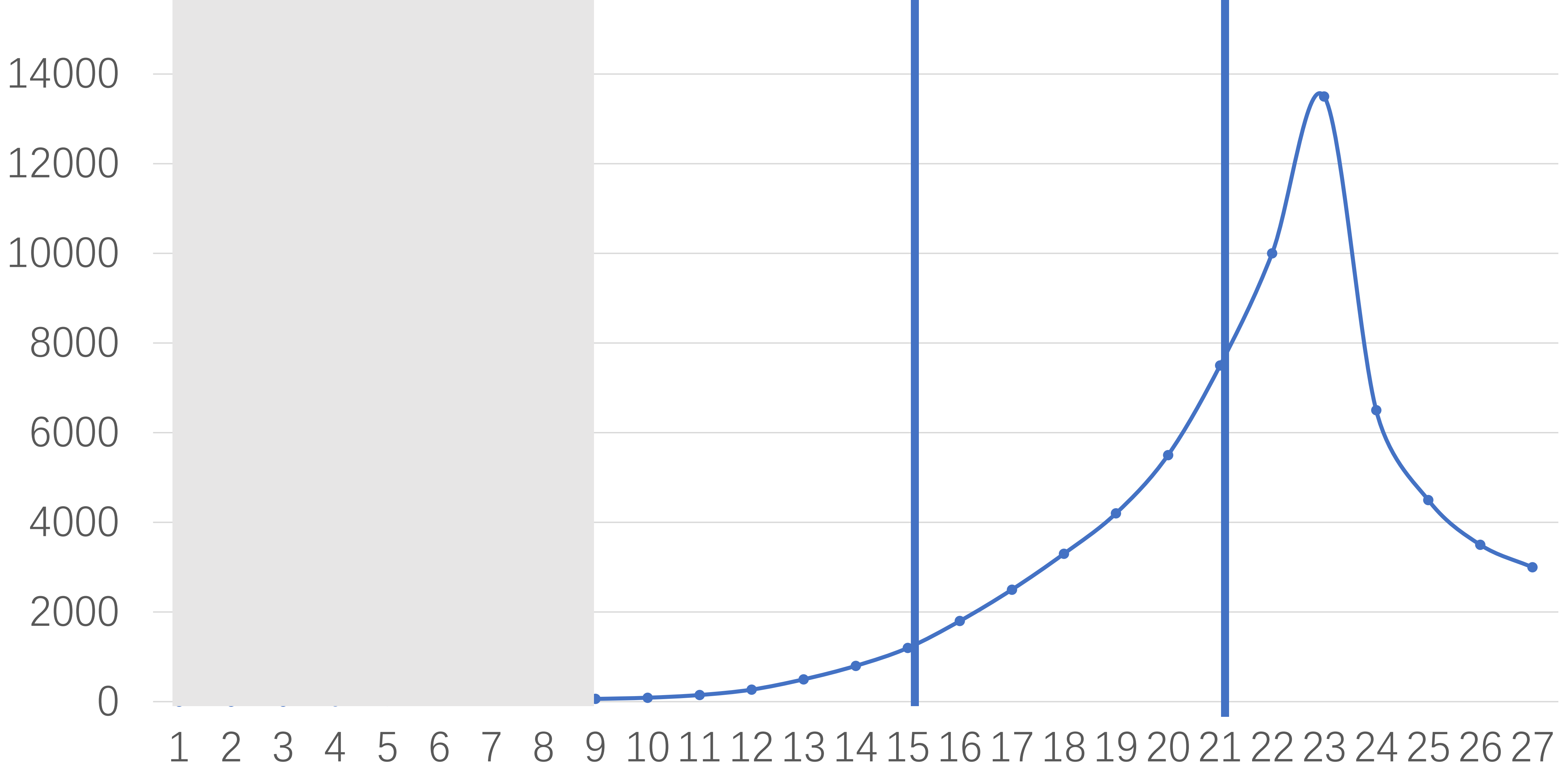
Ternary Search for max

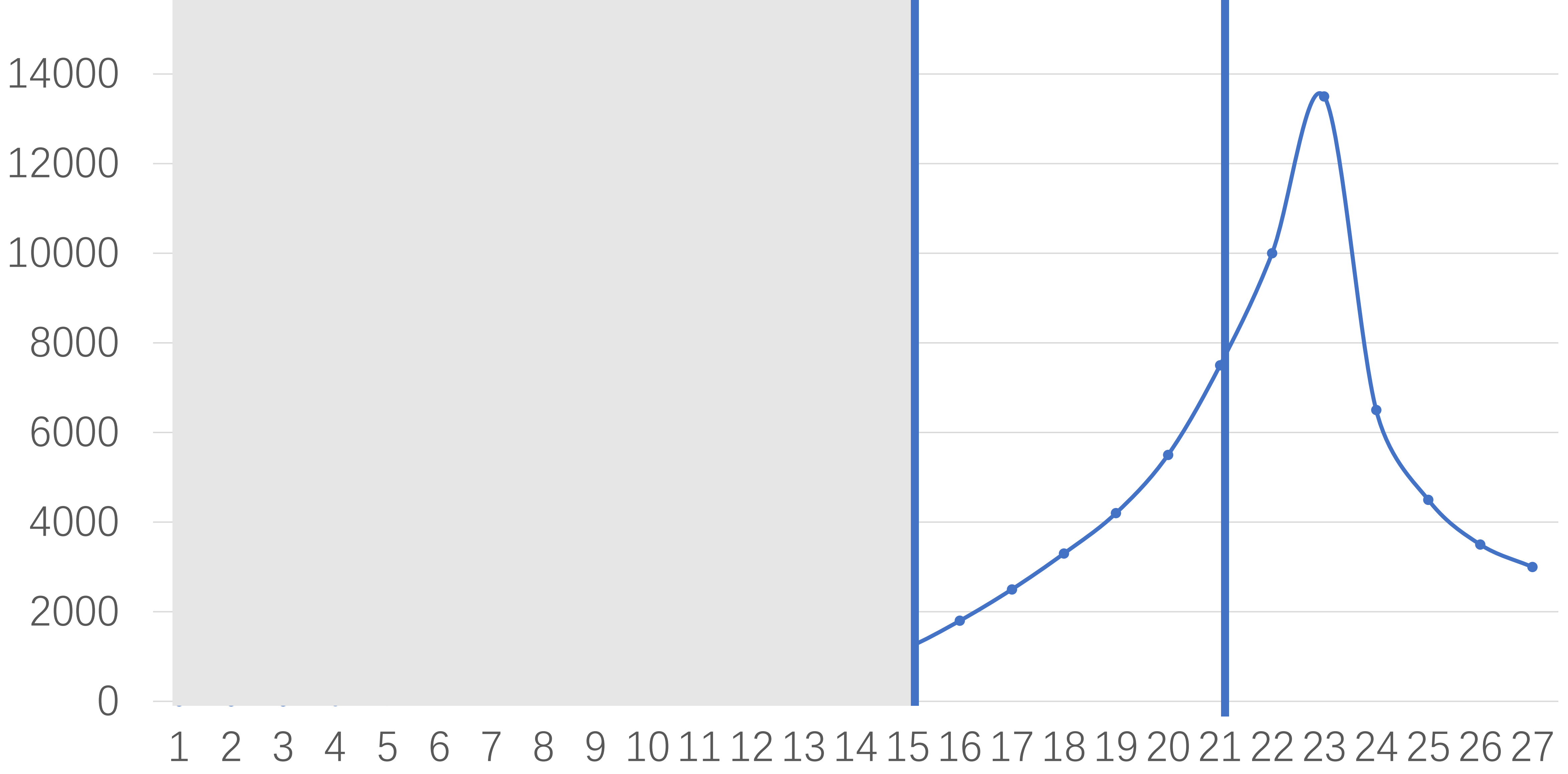
- **When working on a range (l, r)**
- **Find the 1/3-th position and the 2/3-th**
- **Let $l' = l + (r - l)/3$**
- **Let $r' = l + 2(r - l)/3$**
- **If $f(l') < f(r')$:**
  - narrow down the range to (l', r)
- **If $f(l') > f(r)$:**
  - narrow down the range to (l, r')
- **(Drop the part on the smaller side)**
- **When l and r are sufficiently close, report**

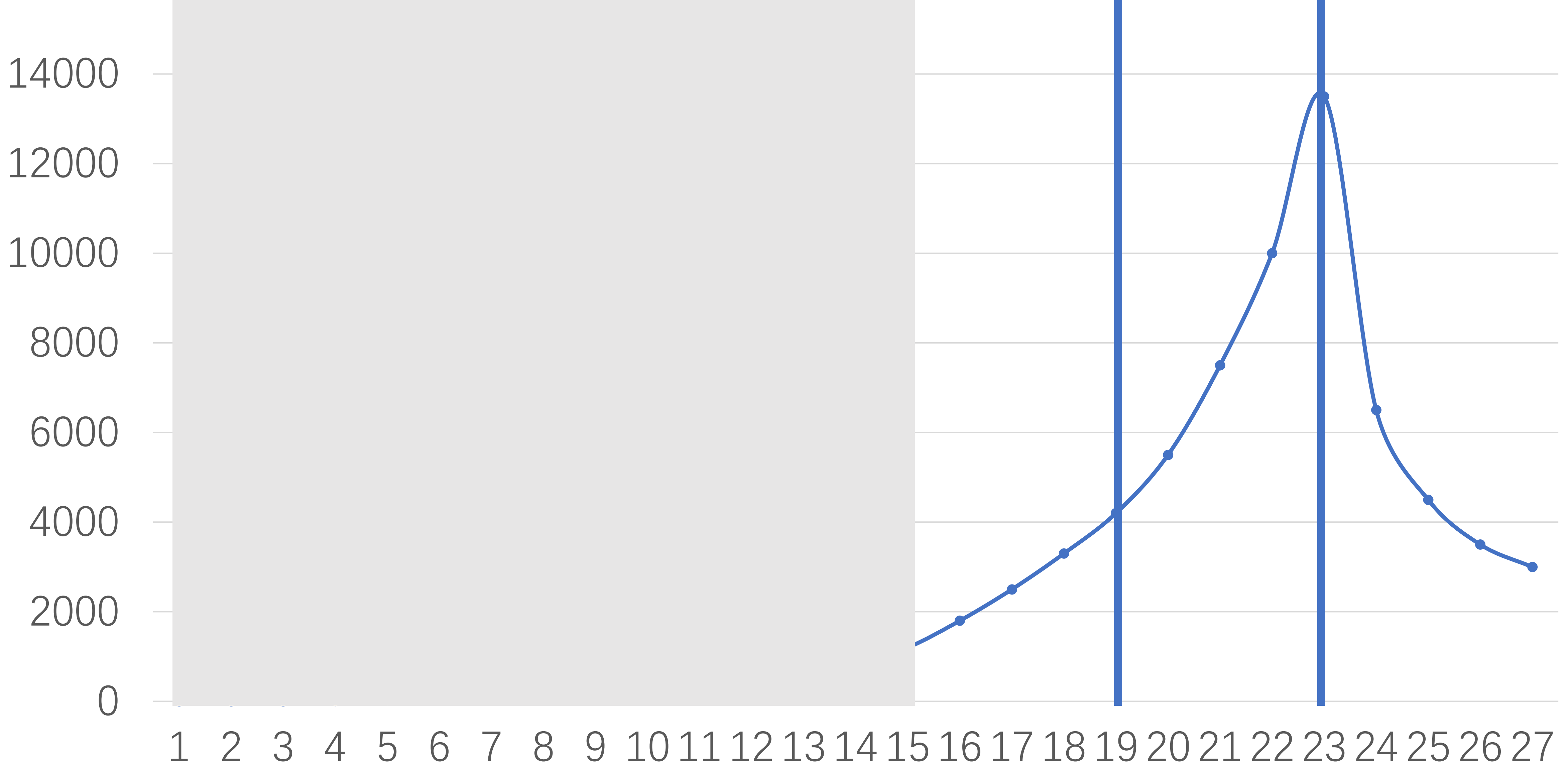


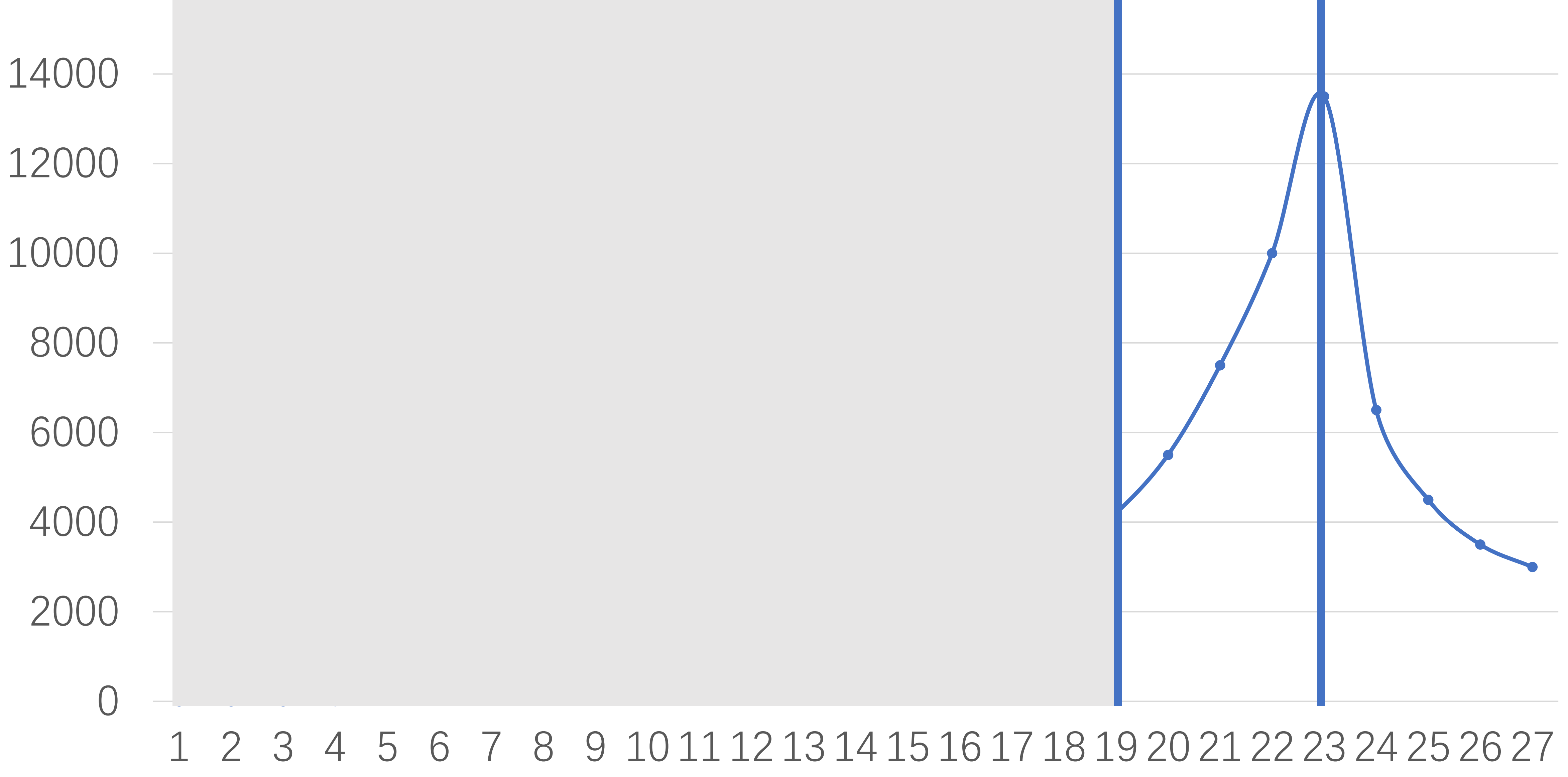

14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

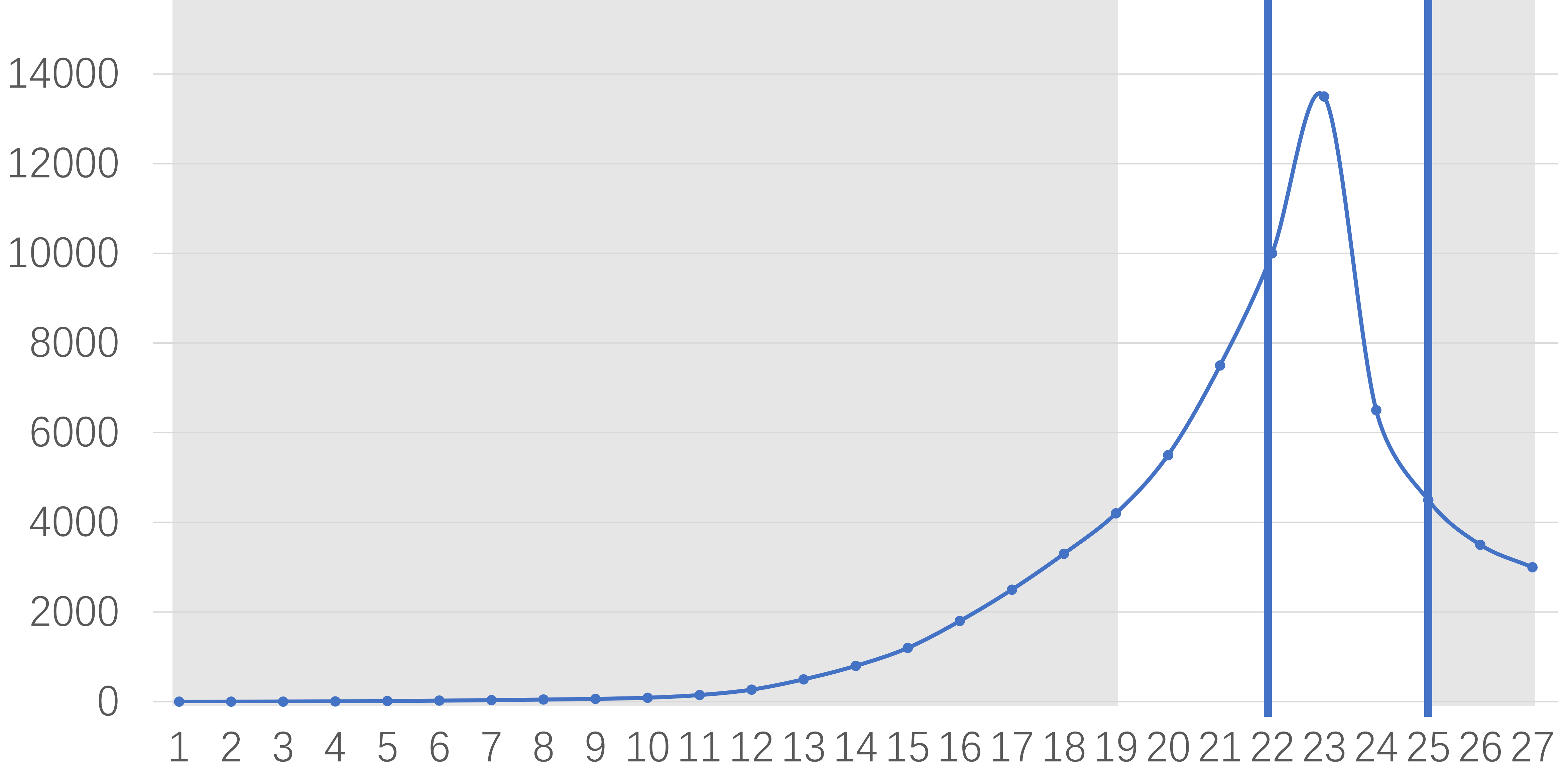

14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

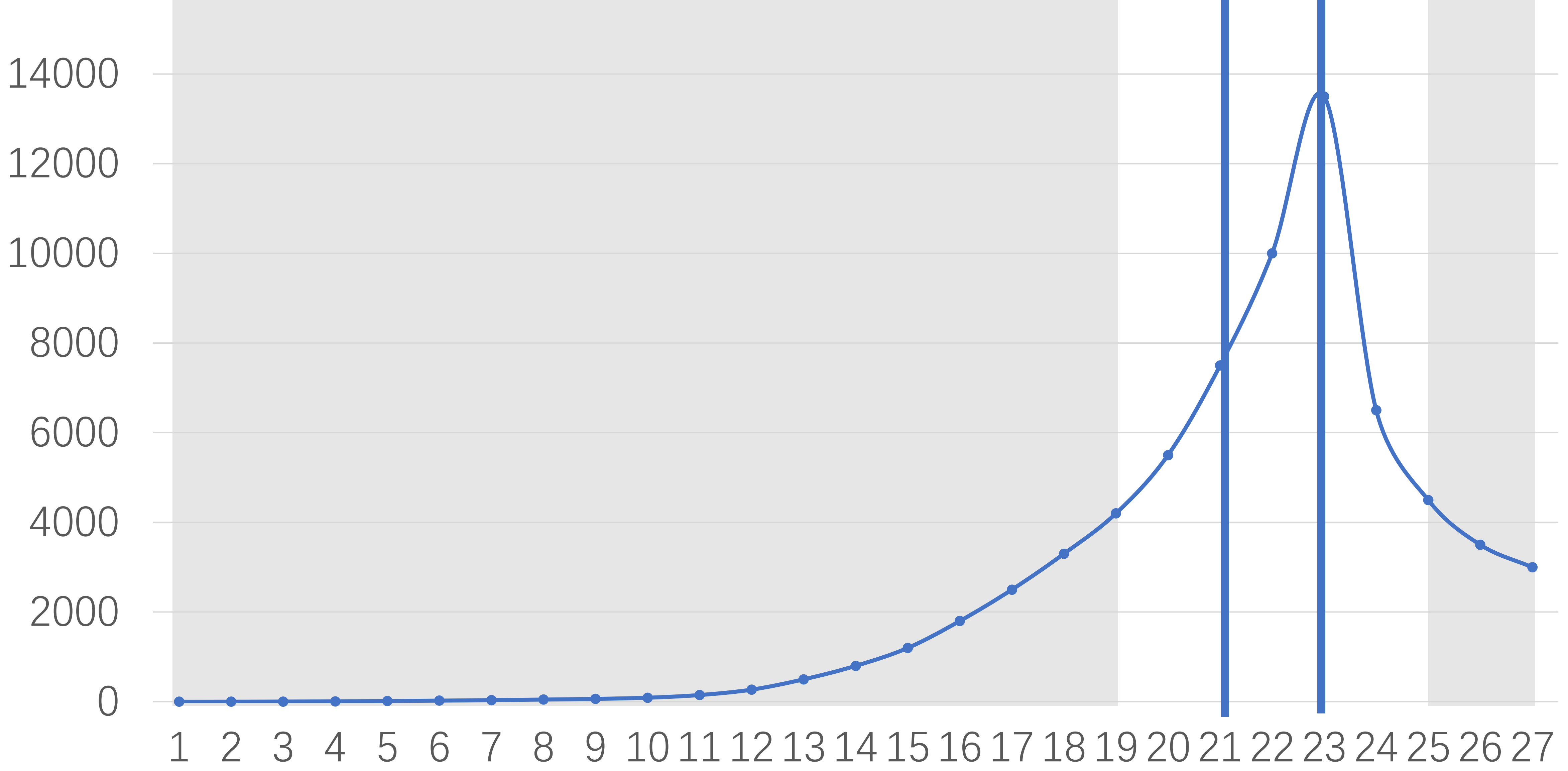

14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27


14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

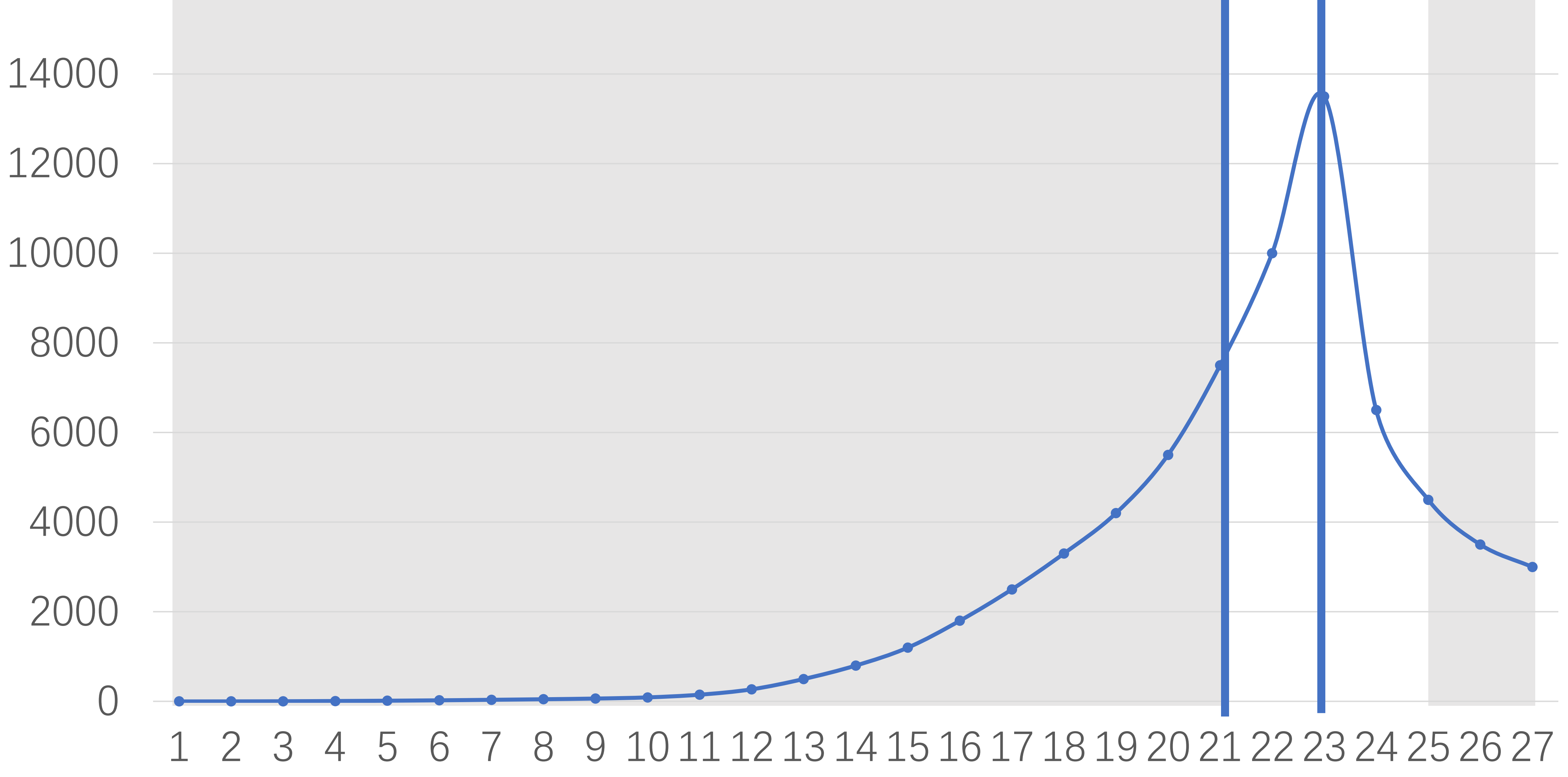

14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

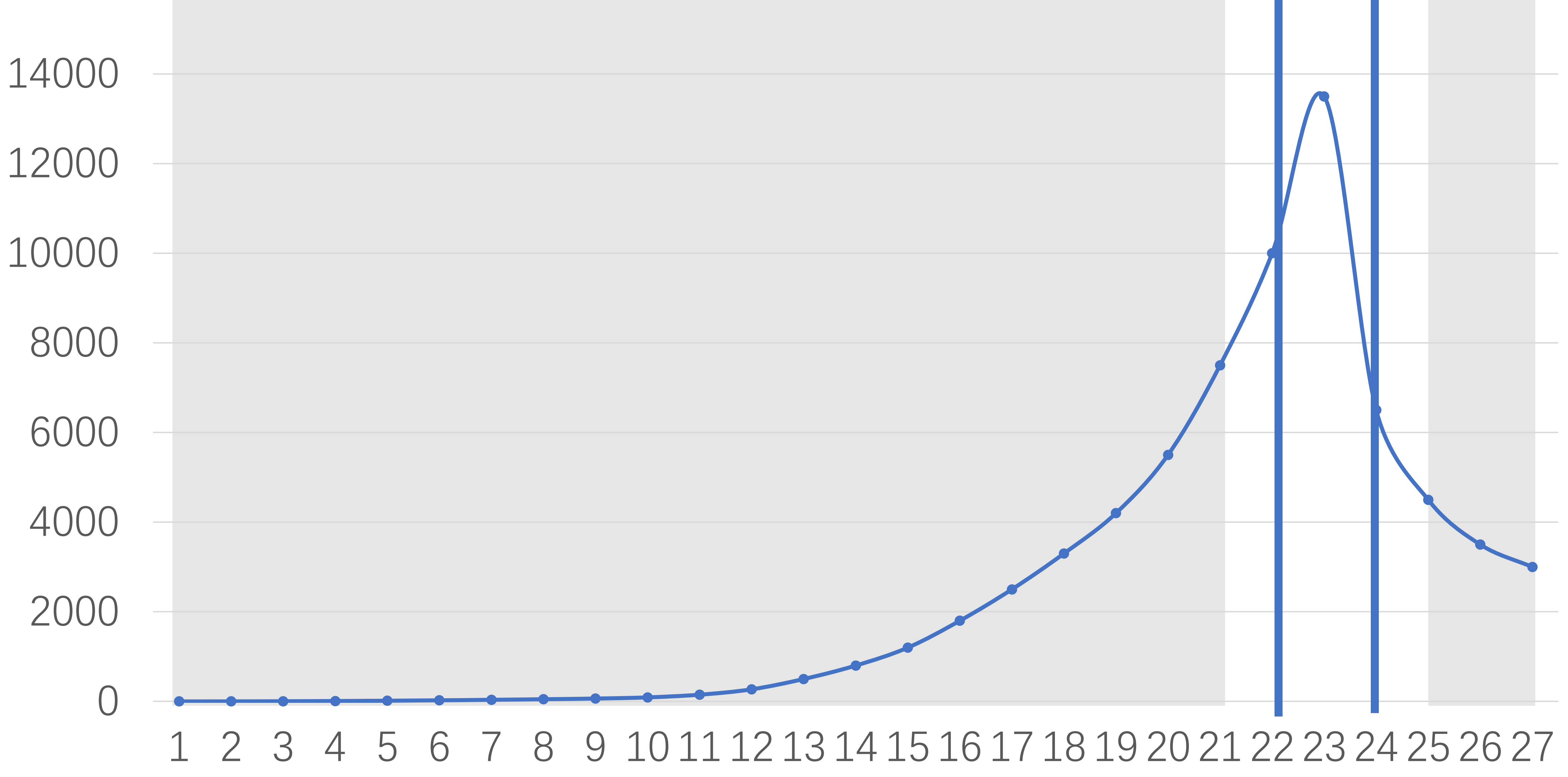

14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

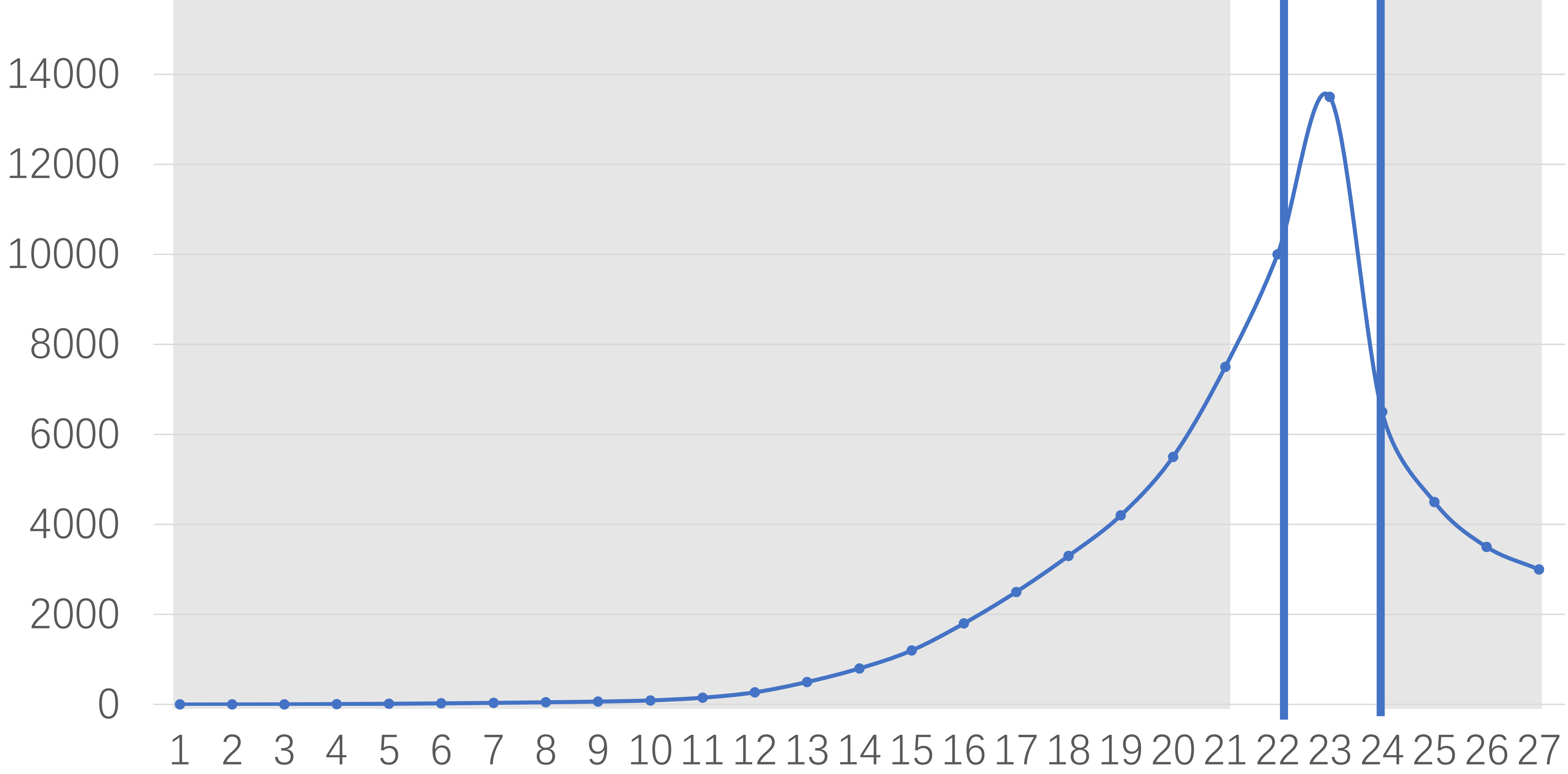

14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

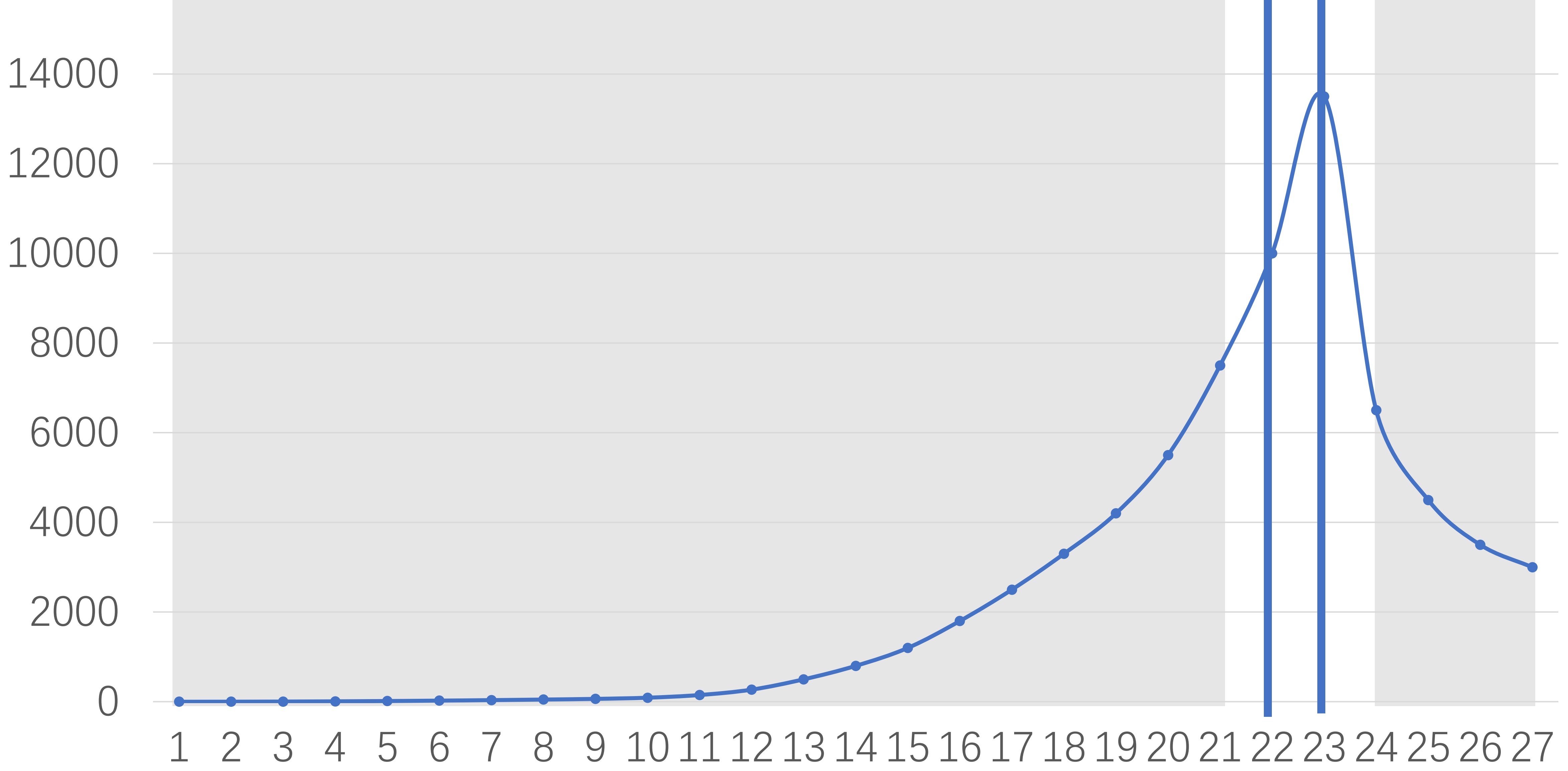


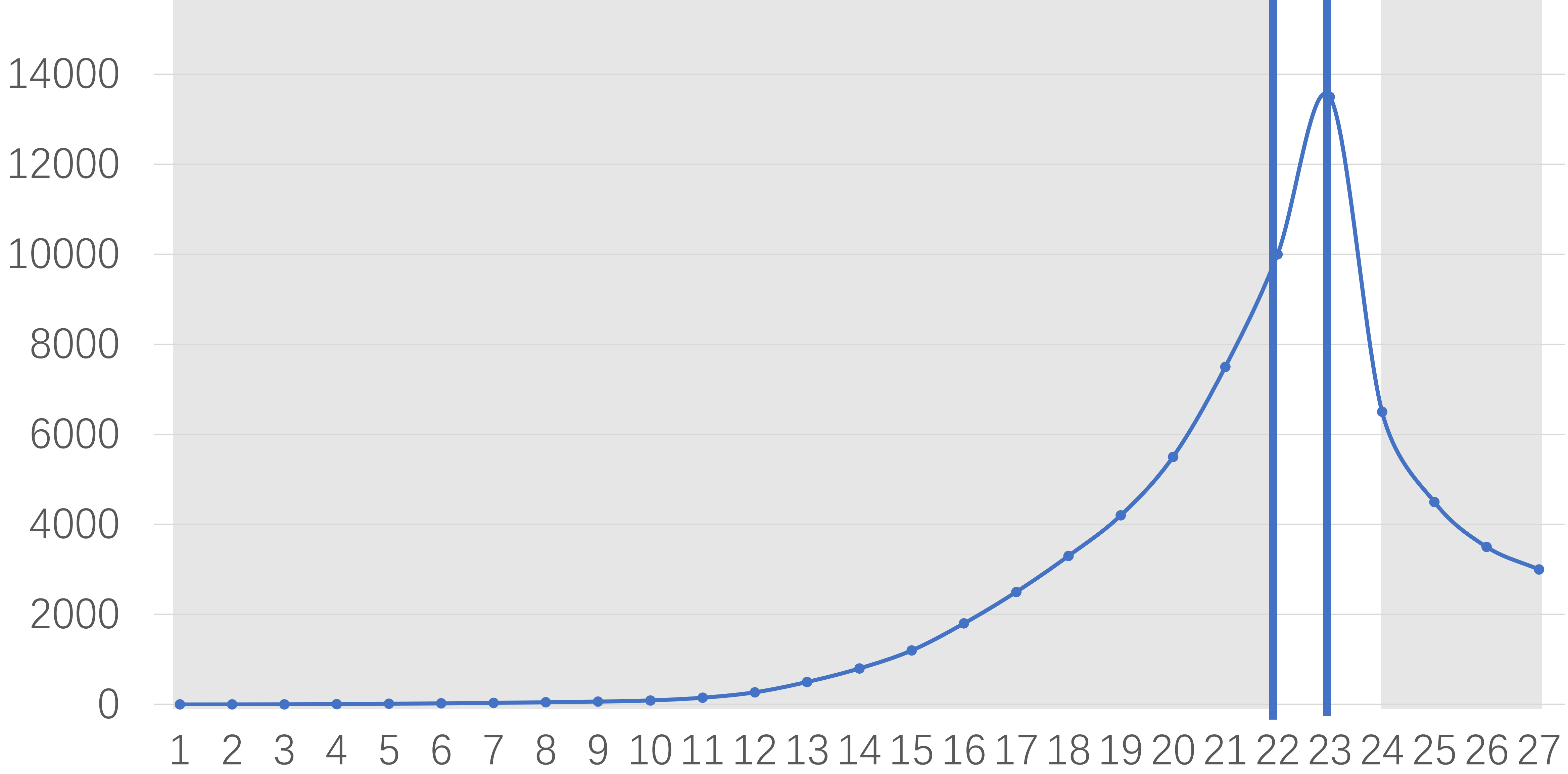
14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27


14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27


14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27


14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27


14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27



If we work on a range of length n, how many tests do we need?

Max at 23!

Ternary Search

- **Exclude 1/3 of the range every test.**
- **O(log n) tests in total. (n is the number of possible answers)**
- **More precisely, it is** $\log_{3/2} n$
- **What if we want to minimize # tests?**
- **0.618-search: choose position at 0.618 and 1-0.618**

Summary for divide-and-conquer

- **Widely used in algorithm design**
  - Classic algorithms: mergesort, quicksort, matrix multiplication
  - Many parallel algorithms
- **Binary search: finding the "zero" for a monotonic function**
  - Also widely used in binary searching the answer and changing an optimizing problem into a checking problem
- **Ternary search: finding the maximum of a unimodal function**

- **Next lecture: Greedy Algorithms**