Introduction to

Algorithm Design and Analysis

[04] QuickSort

Jingwei Xu
https://ics.nju.edu.cn/~xjw/
Institute of Computer Software
Nanjing University

In the Last Class …

- **Recursion in algorithm design**
  - The divide and conquer strategy
  - Proving the correctness of recursive procedures
- **Solving recurrence equations**
  - Some elementary techniques
  - Master theorem

QuickSort

- The sorting problem

- InsertionSort
- Analysis of InsertionSort

- QuickSort
- Analysis of QuickSort

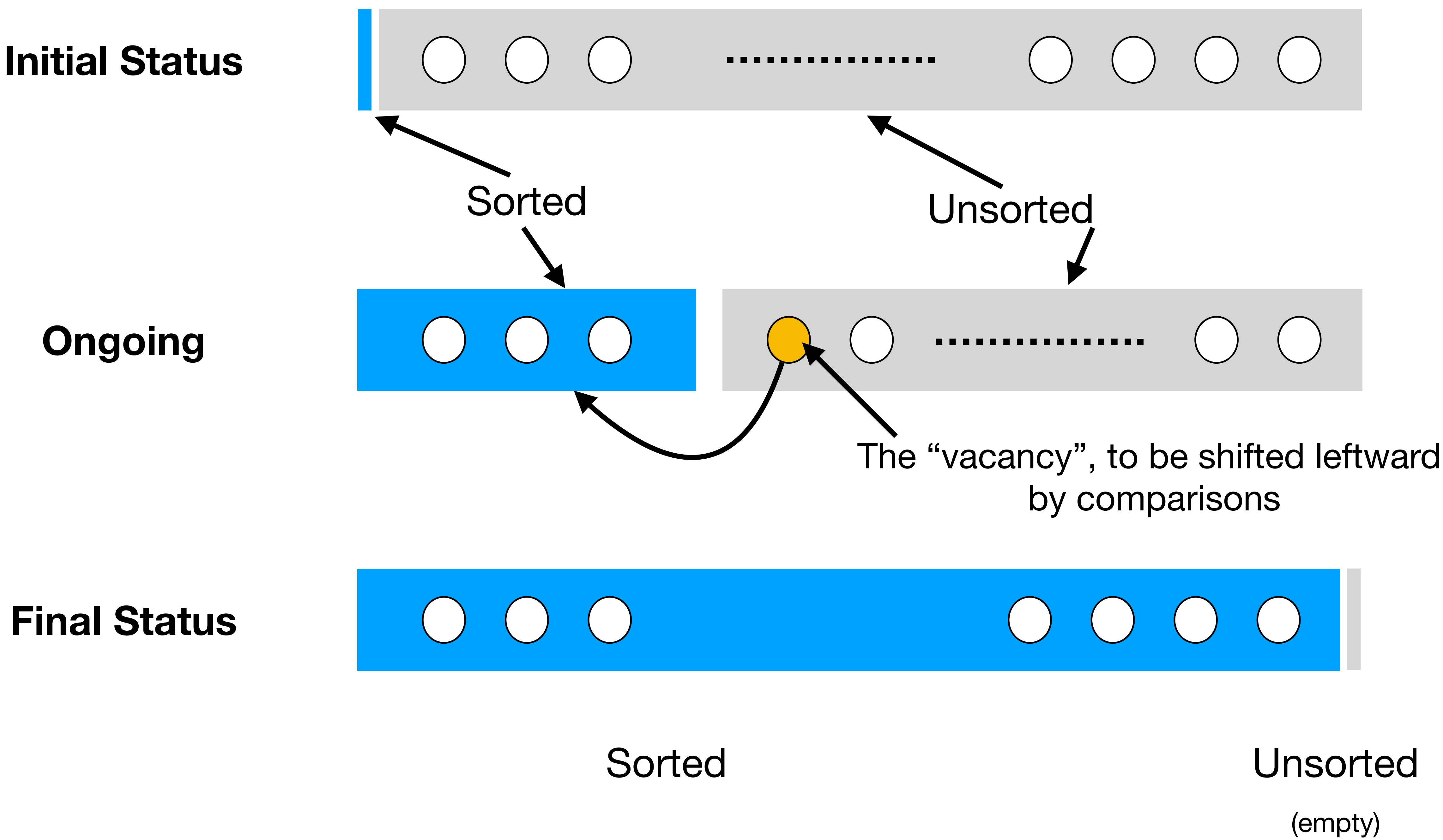
The Sorting Problem

- **Sorting**
  - E.g., sort all the students according to their GPA
- **Assumptions for analysis of sorting**
  - What to sort?
    - Problem size n: elements a1,a2,…,an with no identical keys
  - In which order to sort?
    - Sort in increasing order
  - What are the inputs likely to be?
    - Each possible input appears with the same probability

Comparison-based Sorting

- **Sorting a number of keys**
  - The class of "algorithms that sort by comparison of keys"
- **Critical operation**
  - Comparison between two keys
  - No other operations are allowed for sorting
- **Amount of work done**
  - The number of critical operations (key comparisons)

As Simple as Inserting



Shifting Vacancy

- **int shiftVac(element[] E, int vacant, key x)**
- **Precondition: vacant is nonnegative**
- **Postconditions: Let xLoc be the value returned to the caller, then:**
    - Elements in E at indexes less than xLoc are in their original positions and have keys less than or equal to x.
    - Elements in E at positions (xLoc+1, …, vacant) are greater than x and were shifted up by one position from their positions when shiftVac was invoked.

Shifting Vacancy: Recursion

```
int shiftVacRec(Element[] E, int vacant, key x)
    int xLoc;
    if(vacant == 0)
        xLoc = vacant;
    else if (E[vacant - 1].key ≤ x)
        xLoc = vacant;
    else
        E[vacant] = E[vacant - 1];
        xLoc = shiftVacRec(E, vacant - 1, x);
    return xLoc;
```

Shifting Vacancy: Iteration

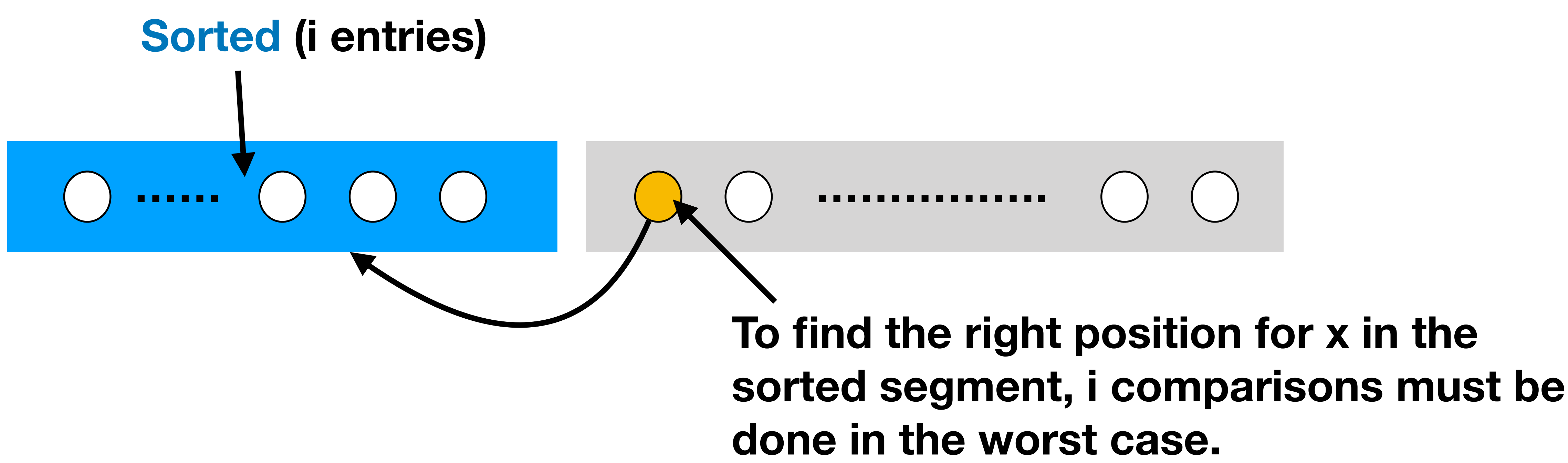
```
int shiftVac(Element[] E, int xindex, key x)
    int vacant, xLoc;
    vacant = xindex;
    xLoc = 0;  //Assume failure
    while(vacant > 0)
        if(E[vacant - 1].key ≤ x)
            xLoc = vacant; //Succeed
            break;
        E[vacant] = E[vacant - 1];
        vacant -= 1; //Keep looking
    return xLoc;
```

InsertionSort: the Algorithm

- **Input: E(array), n ≥ 0(size of E)**
- **Output: E, ordered non-decreasingly by keys**
- **Procedure:**

```
void InsertionSort(Element[] E, int n)
    int xindex;
    for(xindex = 1; xindex < n; xindex ++){
        element current = E[xindex];
        Key x = current.key;
        int xLoc = shiftVac(E, xindex, x);
        E[xLoc] = current;
  return;
```

Worst-Case Analysis



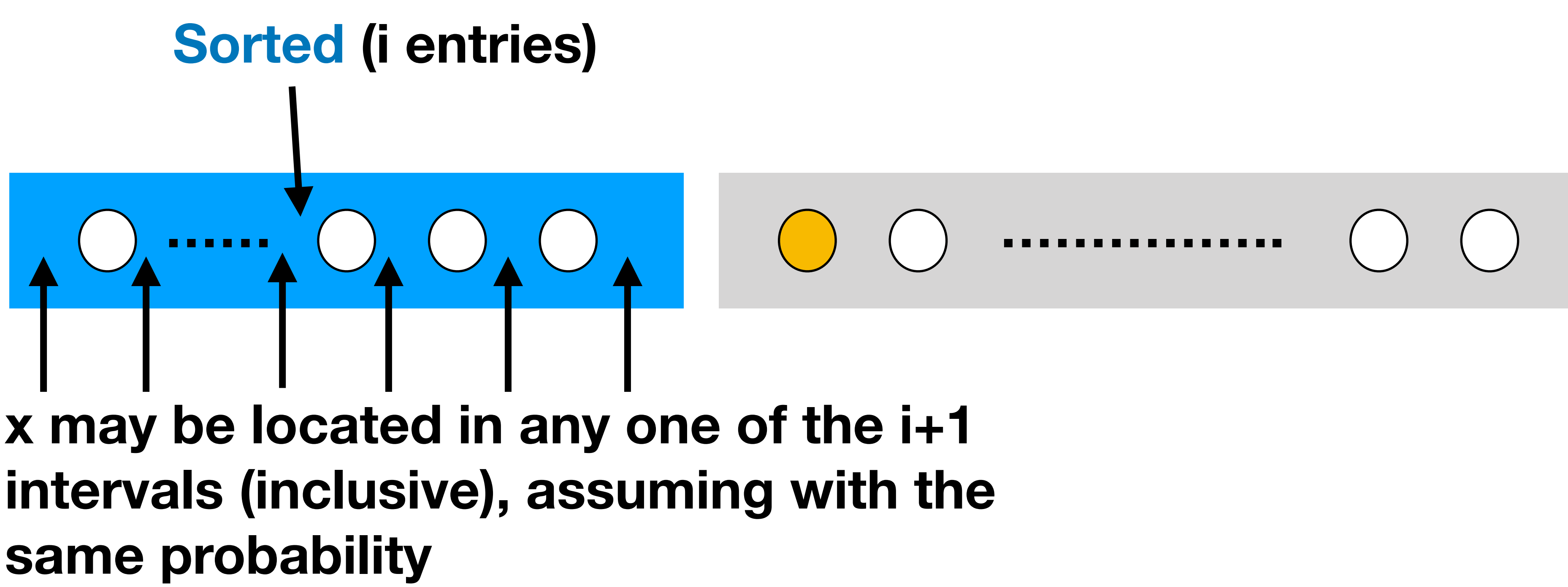
- At the beginning, there are n-1 entries in the unsorted segment, so:

$$W(n) \leq \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

The input for which the upper bound is reached does exist, so:

$$W(n) = \Theta(n^2)$$

Average-Case Behavior



- Assumptions:
  - All permutations of the keys are equally likely as input.
  - There are not different entries with the same keys.

Note: For the i-th and (i+1)-th intervals (leftmost), only one comparisons is needed.

Average Complexity

- The expected number of comparisons in shiftVac to find the location for the (i+1)-th element:

$$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{1}{i+1}(i) = \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$$

- For all n-1 insertions:

$$A(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n}\frac{1}{i}$$

$$= \frac{n(n-1)}{4} + n - \sum_{j=1}^{n}\frac{1}{j} = \frac{n^2}{4} + \frac{3n}{4} - \ln n \in \Theta(n^2)$$

Inversion and Sorting

- **An unsorted sequence E:**
  - $\{x_1, x_2, x_3, \ldots, x_{n-1}, x_n\} = \{1, 2, 3, \ldots, n-1, n\}$
- **$\langle x_i, x_j \rangle$ is an inversion if $x_i > x_j$, but $i < j$**
- **Sorting $\equiv$ Eliminating inversions**
  - All the inversions must be eliminated during the process of sorting

Eliminating Inverses: Worst Case

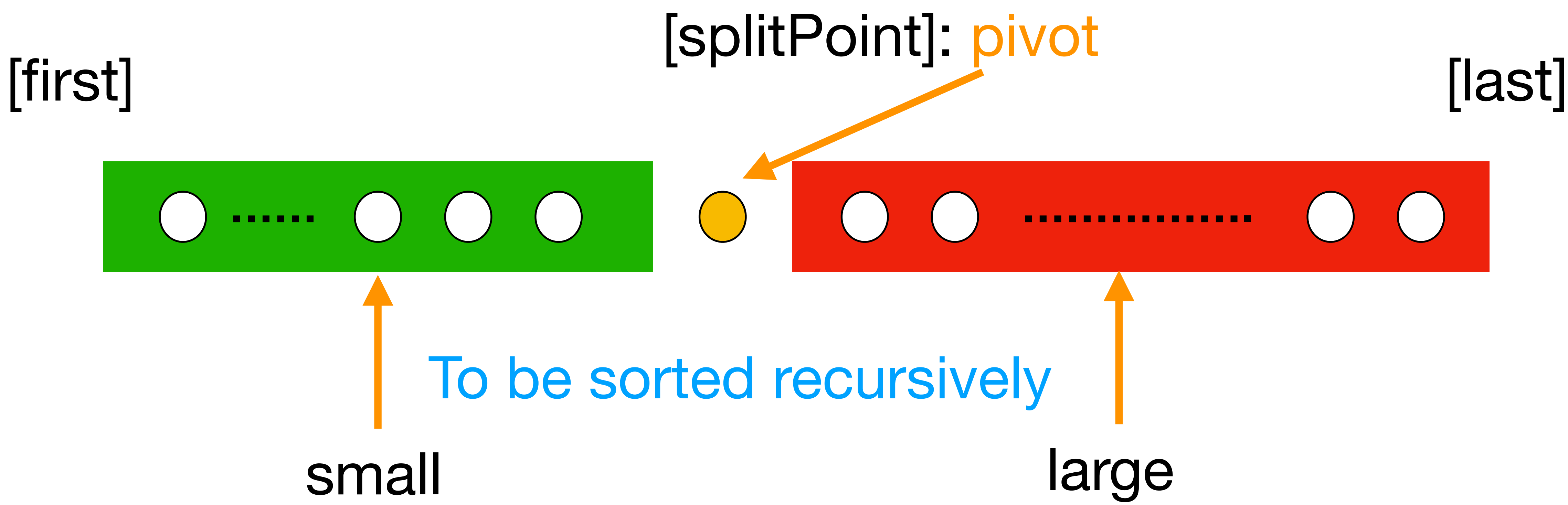
- Local comparison is done between two adjacent elements
- At most one inversion is removed by a local comparison
- There do exist inputs with $n(n-1)/2$ inversions, such as (n, n-1, …, 3, 2, 1)
- The worst-case behavior of any sorting algorithm that remove at most one inversion per key comparison must in $\Omega(n^2)$

Eliminating Inversions: Average Case

- **Computing the average number of inversions in inputs of size n (n > 1):**
  - Transpose: $x_1, x_2, x_3, \ldots, x_{n-1}, x_n$ => $x_n, x_{n-1}, \ldots x_3, x_2, x_1$
  - For any i, j, $(1 \le j \le i \le n)$, the inversion (x_i, x_j) is in exactly one sequence in a transpose pair.
  - The number of inversions (x_i, x_j) on n distinct integers is $n(n-1)/2$.
  - So, the average number of inversions in all possible inputs is $n(n-1)/4$, since exactly $n(n-1)/2$ inversions appear in each transpose pair.
- **The average behavior of any sorting algorithm that remove at most one inversion per key comparison must in $\Omega(n^2)$.**

QuickSort: the Strategy

- **Divide the array to be sorted into two parts: “small” and “large”, which will be sorted recursively.**



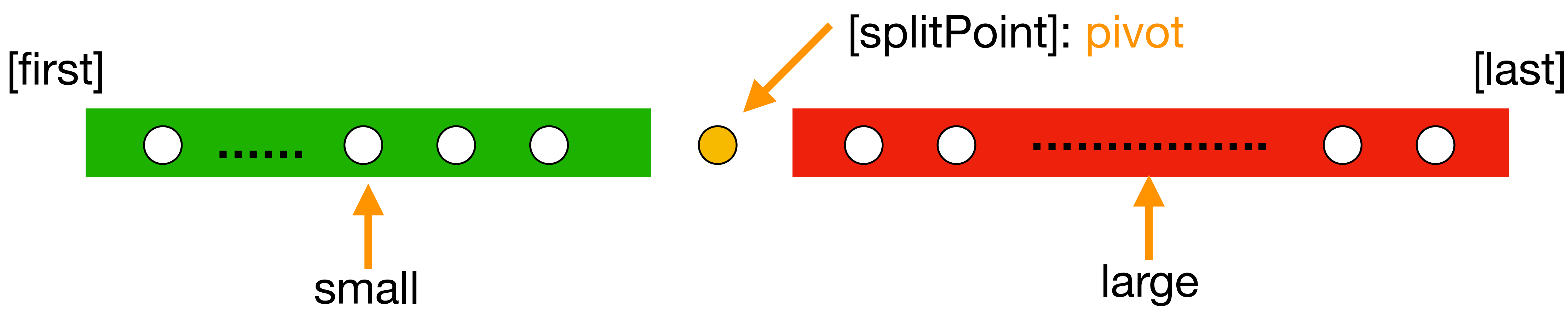
for any element in this segment, the key is less than pivot.

for any element in this segment, the key is not less than pivot.

QuickSort: the Strategy

- **Divide**
  - “small” and “large”
- **Conquer**
  - Sort “small” and “large” recursively
- **Combine**
  - Easily combine sorted sub-array

Hard divide,
easy combination



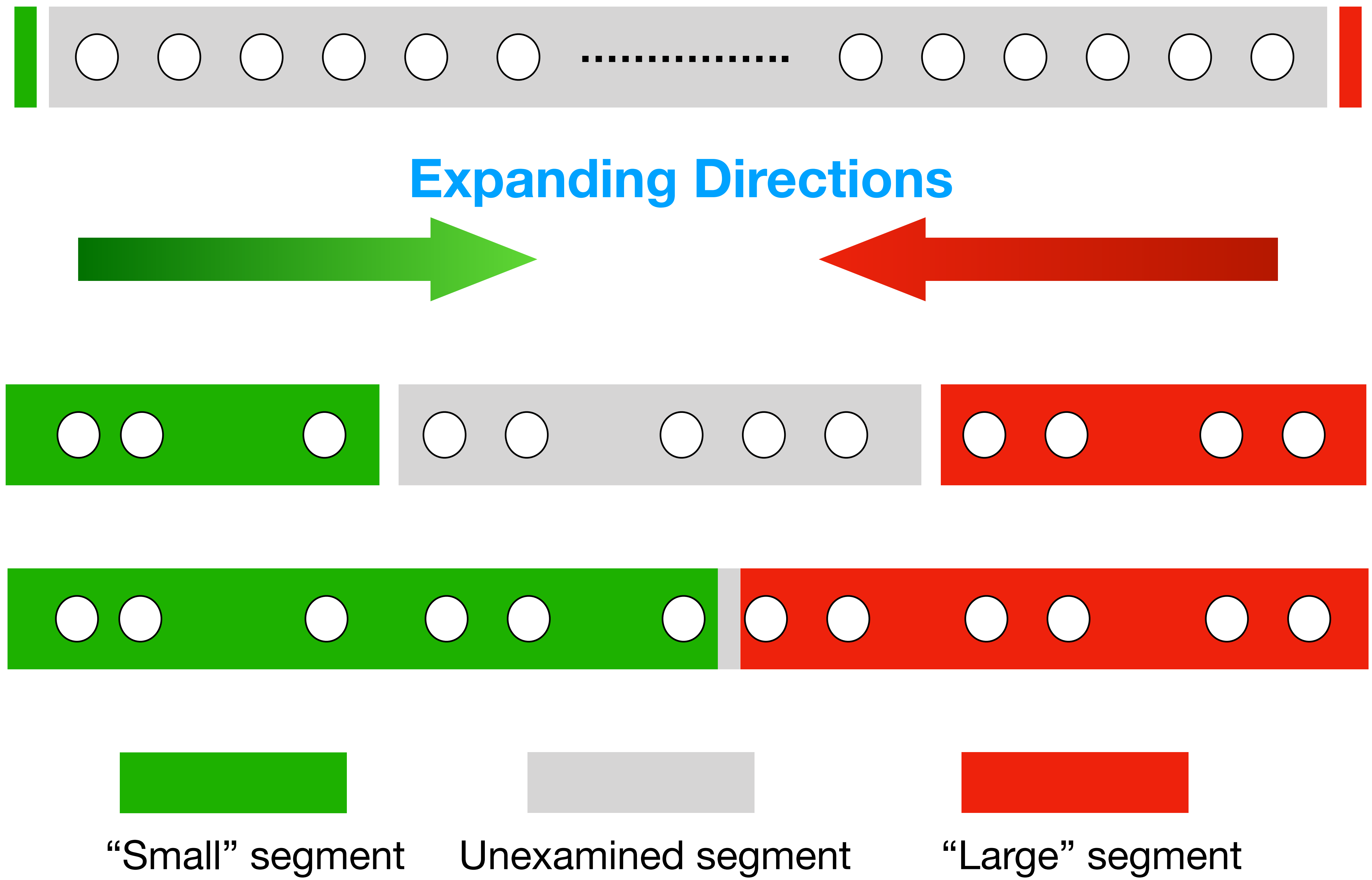
QuickSort: the Algorithm

- **Input: Array E, indexes first, and last, such that elements E[i] are defined for first≤i≤last.**
- **Output: E[first], …, E[last] is a sorted rearrangement of the same elements.**
- **The procedure:**

```
void quickSort(Element[] E, int first, int last)
    if(first < last)
        Element pivotElement = E[first];
        Key pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first, last);
        E[splitPoint] = pivotElement;
        quickSort(E, first, splitPoint - 1);
        quickSort(E, splitPoint - 1, last);
    return;
```

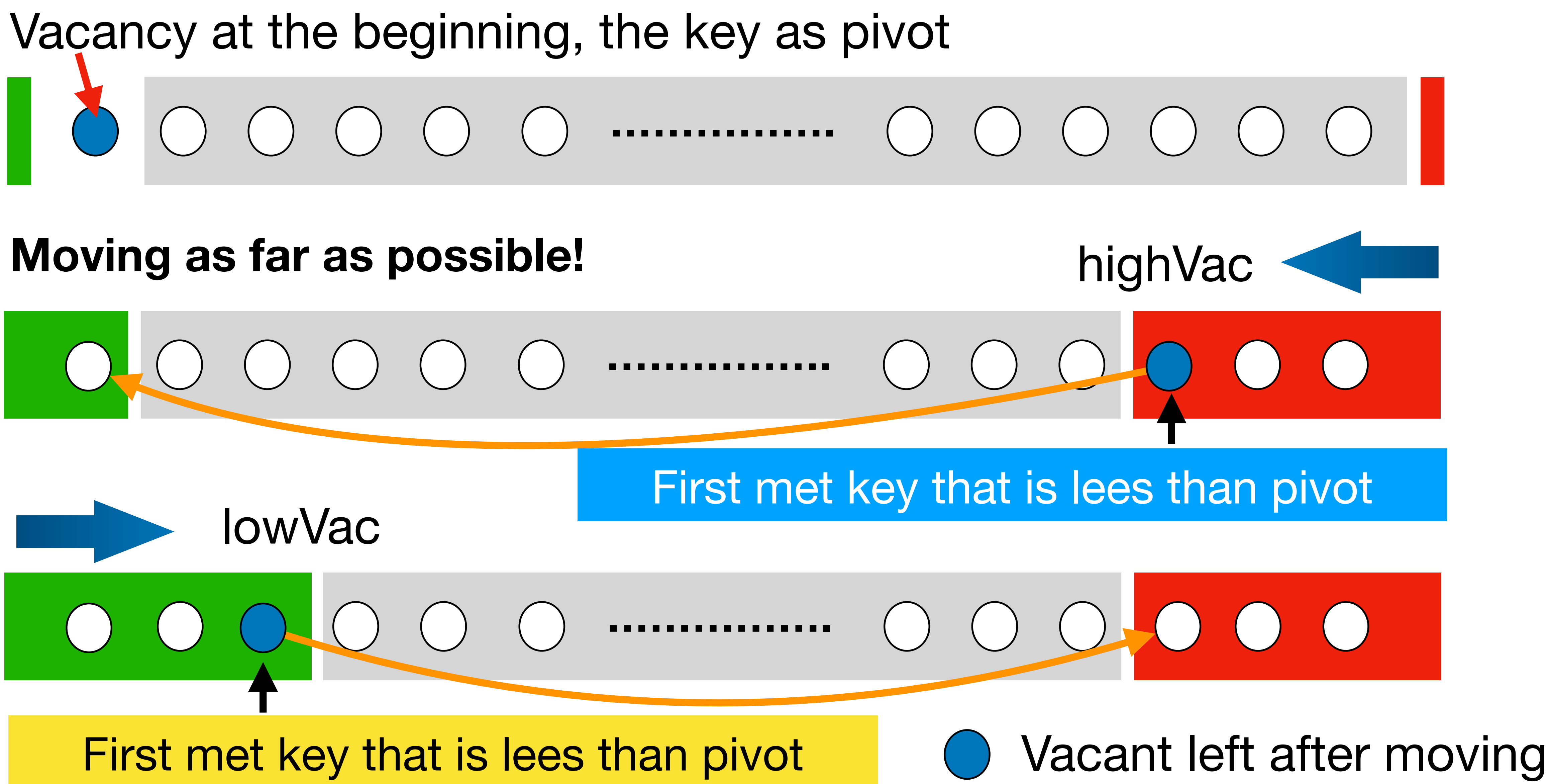
The splitting point is chosen arbitrarily, as the first element in the array segment here.

Partition: the Strategy



Partition: the Process

- **Always keep a vacancy before completion**



Partition: the Algorithm

- **Input: Array E, pivot, the key around which to partition, and indexes first, and last, such that elements E[i] are defined for first+1≤i≤last and E[first] is vacant. It is assumed that first<last.**
- **Output: Returning splitPoint, the elements originally in first+1, …, last are rearranged into two subranges, such that**
  - the keys of E[first], …, E[splitPoint - 1] are less than pivot, and
  - the keys of E[splitPoint + 1], …, E[last] are not less than pivot, and
  - first≤splitPoint≤last, and E[splitPoint] is vacant.

Partition: the Procedure

```
int partition(Element[ ] E, Key pivot, int first, int last)
    int low, high;
low = first; high = last;
while(low < high){
    int highVac =
        extendLargeRegion(E, pivot, low, high);
    int lowVac =
        extendSmallRegion(E, pivot, low + 1, highVac);
    low = lowVac; high = highVac - 1;
}
return low; // this is the splitPoint
```

highVac has been filled now

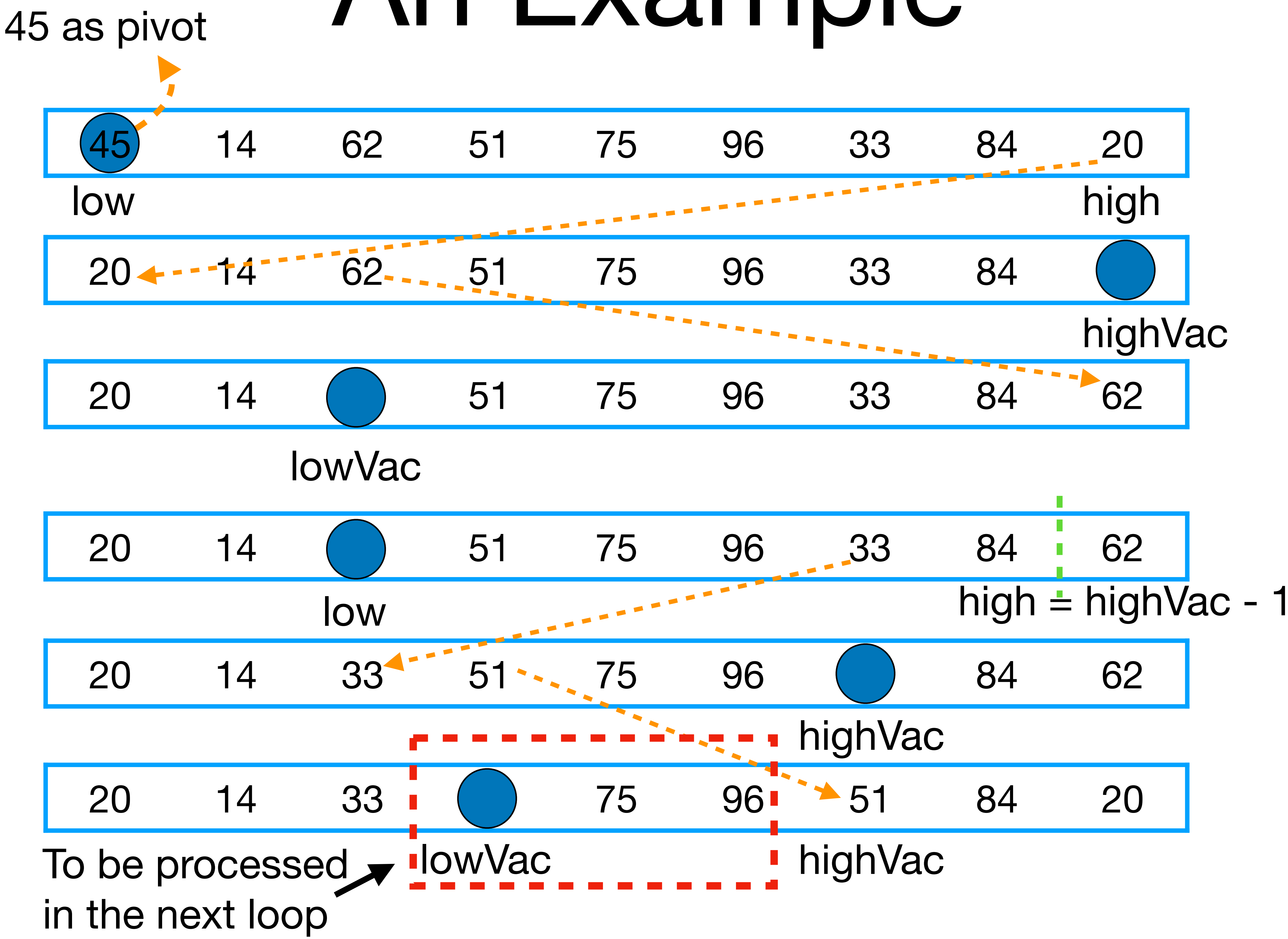
Extending Regions

- **Specification for**

  extendLargeRegion(Element[ ] E, Key pivot, **int** lowVac, **int** high)

  - Precondition:
    - lowVac < high
  - Postcondition:
    - if there are elements in E[lowVac + 1], …, E[high] whose key is less than pivot, then the rightmost of them is moved to E[lowVac], and its original index is returned.
    - If there is no such element, lowVac is returned;

An Example


45 as pivot
45 14 62 51 75 96 33 84 20
low
high
20 14 62 51 75 96 33 84
highVac
20 14 51 75 96 33 84 62
lowVac
20 14 51 75 96 33 84 62
low
high = highVac - 1
20 14 33 51 75 96 84 62
highVac
20 14 33 75 96 51 84 20
lowVac
highVac
To be processed
in the next loop

Worst Case: a Paradox

- **For a range of k positions, k-1 keys are compared with the pivot (one is vacant).**
  - If the pivot is the smallest, then the “large” segment has all the remaining k-1 elements, and the “small” segment is empty.
  - If the elements in the array to be sorted has already in ascending order (the Goal), then the number of comparison that Partition has to do is:

$$\sum_{k=2}^{n}(k-1)=\frac{n(n-1)}{2}\in O(n^2)$$

Average-case Analysis

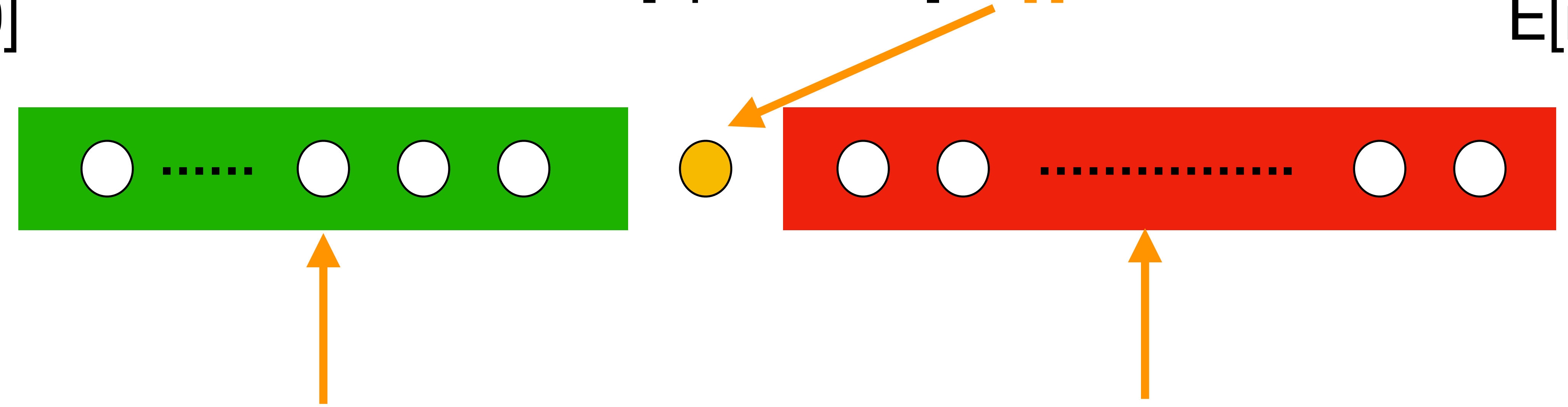
- **Assumption: all permutation of the keys are equally likely.**
- **A(n) is the average number of key comparisons done for range of size n.**
    - In the first cycle of Partition, n-1 comparisons are done.
    - If split point is E[i] (each i has probability 1/n), Partition is to be executed recursively on the subrange [0, …, i-1] and [i+1, …, n-1]

The Recurrence Equation

[splitPoint]: E[i]

E[0]

E[n-1]



subrange 1: size = i

subrange 2: size = n-1-i

why the assumed probability still holds for each subrange?

with i∈{0, 1, 2,…, n-1}, each value with the probability 1/n

the average number of key comparison A(n) is:

$$A(n) = (n-1) + \sum_{i=0}^{n-1} \frac{1}{n}[A(i) + A(n-1-i)] \quad \text{for } n \geq 2$$

A(1)=A(0)=0

The number of key comparison in the first cycle (finding the splitPoint) is n-1

Simplified Recurrence Equation

- **Note:** $\displaystyle\sum_{i=0}^{n-1} A(i) = \sum_{i=0}^{n-1} A[(n-1)-i]$ $\quad A(0) = 0$

- **So:** $\displaystyle A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$ for n ≥ 1

- **Two approaches to solve the equation**
  - Guess, and prove by induction
  - Solve directly

Guess the Solution

- **A special case as the clue for a smart guess**
  - Assuming that Partition divide the problem range into 2 subranges of about the same size.
  - So, the number of comparison Q(n) satisfy:

$$Q(n) \approx n + 2Q(n/2)$$

  - Applying Master Theorem, cases:

$$Q(n) \in \Theta(n \log n)$$

Note: here, b=c=2, so E=log(b)/log(c)=1, and, $f(n)=n^E=n$

Inductive Proof: A(n)∈O($n$ln$n$)

- **Theorem: A(n)≤c$n$ln$n$ for some constant c, with A(n) defined by the recurrence equation above.**
- **Proof:**
  - By induction on n, the number of elements to be sorted. Base case (n=1) is trivial.
  - Inductive assumption: $A(i) \leq ci \ln i$ for 1≤i<n

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \leq (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} ci \ln i$$

Note: $\frac{2}{n}\sum_{i=1}^{n-1} ci \ln i \leq \frac{2c}{n}\int_1^n x \ln x dx \approx \frac{2c}{n}\left(\frac{n^2 \ln n}{2} - \frac{n^2}{4}\right) = cn \ln n - \frac{cn}{2}$

So, $A(n) \leq cn \ln n + n\left(1 - \frac{c}{2}\right) - 1$

Let c = 2, we have $A(n) \leq 2n \ln n$

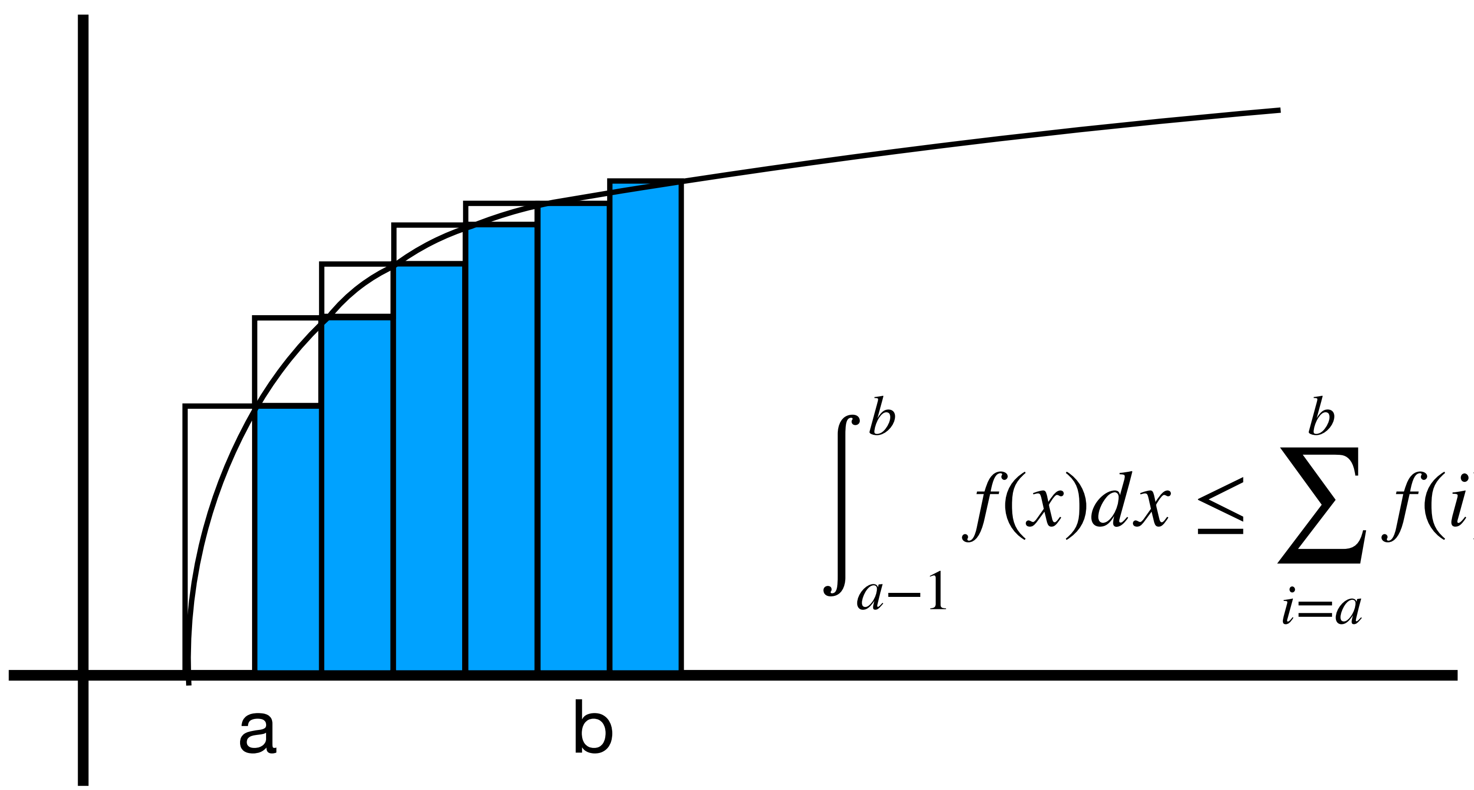
For Your Reference

$$\int_1^n x^k \ln x dx = \left(\frac{x^{k+1} \ln x}{k+1} - \frac{x^{k+1}}{(k+1)^2} \right) \Bigg|_1^n$$

$$= \frac{n^{k+1} \ln n}{k+1} - \frac{n^{k+1}}{(k+1)^2} + \frac{1}{(k+1)^2}$$

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + 0.577$$

Harmonic Series



$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx$$

Inductive Proof: A(n)∈Ω(*n*ln*n*)

- **Theorem: A(n)≥c*n*ln*n* for some constant c, with large n**
- **Inductive reasoning:**

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \geq (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} ci\ln i$$

$$= (n-1) + \frac{2c}{n}\sum_{i=2}^{n} i\ln i - 2c\ln n \geq (n-1) + \frac{2c}{n}\int_1^n x\ln x dx - 2c\ln n$$

$$\approx cn\ln n + [(n-1) - c(\frac{n}{2} + 2\ln n)]$$

Let $c < \dfrac{n-1}{\frac{n}{2} + 2\ln n}$, then $A(n) > cn\ln n$ (Note: $\lim_{n\to\infty} \dfrac{n-1}{\frac{n}{2} + 2\ln n} = 2$)

Directly Derived Recurrence Equation

We have $A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$ and

$$A(n-1) = (n-2) + \frac{2}{n-1}\sum_{i=1}^{n-2} A(i)$$

Combining the 2 equations in some way, we can remove all A(i) for i=1, 2, …, n-2

$$\begin{aligned} &nA(n) - (n-1)A(n-1) \\ &= n(n-1) + 2\sum_{i=1}^{n-1} A(i) - (n-1)(n-2) - 2\sum_{i=1}^{n-2} A(i) \\ &= 2A(n-1) + 2(n-1) \end{aligned}$$

So, $nA(n) = (n+1)A(n-1) + 2(n-1)$

Solve the Equation

$$nA(n) = (n+1)A(n-1) + 2(n-1) \Rightarrow \frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

- **We have:** $B(n) = B(n-1) + \dfrac{2(n-1)}{n(n+1)}$ $\quad B(1) = 0$
  - Thus: $B(n) = O(\log n)$

$$\begin{aligned}
B(n) &= \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\sum_{i=1}^{n} \frac{(i+1)-2}{i(i+1)} \\
&= 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)} = 4\sum_{i=1}^{n} \frac{1}{i+1} - 2\sum_{i=1}^{n} \frac{1}{i} \\
&= 4\sum_{i=2}^{n+1} \frac{1}{i} - 2\sum_{i=1}^{n} \frac{1}{i} = 2\sum_{i=1}^{n} \frac{1}{i} - \frac{4n}{n+1} \\
&= O(\log n)
\end{aligned}$$

- **Finally we get**
  - $A(n) = O(n \log n)$

Space Complexity

- **Good news:**
  - Partition is in-place
- **Bad news:**
  - In the worst case, the depth of recursion will be n-1
  - So, the largest size of the recursion stack will be in $\Theta(n)$

More than Sorting

- **QuickSort Partition**
  - O(n)
- **Bolts and nuts**
  - O(nlogn)
- **k-Sorted**
  - O(nlogk)

Thank you!
Q & A