



Introduction to

Algorithm Design and Analysis

[4] QuickSort



Yu Huang

<http://cs.nju.edu.cn/yuhuang>
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 a_1, a_2, \dots, a_n 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

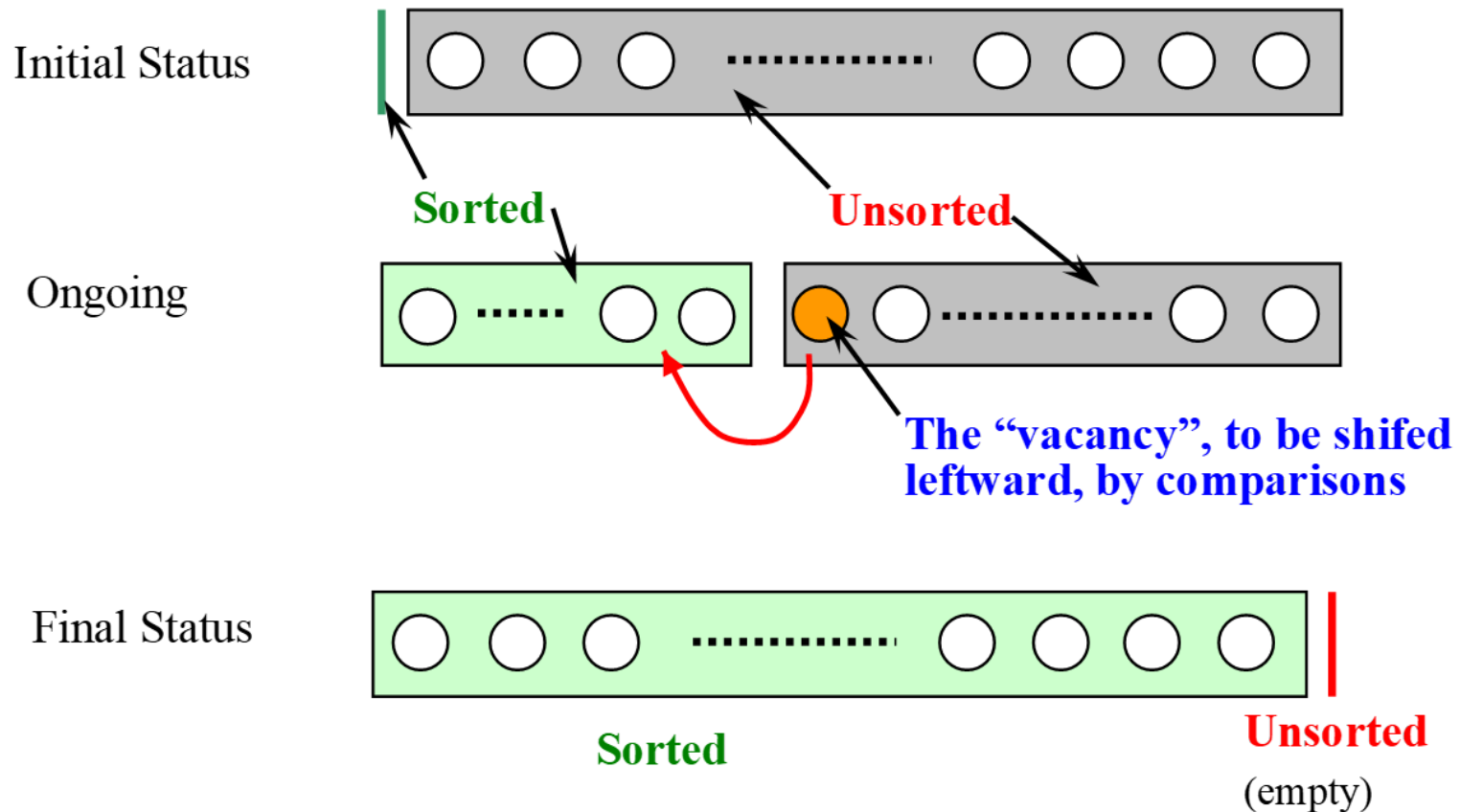


Are the assumptions restrictive?

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

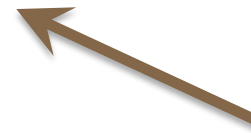
```
int shiftVacRec(Element[] E, int vacant, Key x)
```

```
    int xLoc
```

1. if (vacant==0)
2. xLoc=vacant;
3. else if (E[vacant-1].key≤x)
4. xLoc=vacant;
5. else
6. E[vacant]=E[vacant-1];
7. xLoc=shiftVacRec(E,vacant-1,x);
8. Return xLoc

The recursive call is working on a smaller range, so terminating;

The second argument is non-negative, so precondition holding



Worse case frame stack size is $O(n)$

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--; //Keep Looking
    return xLoc
```



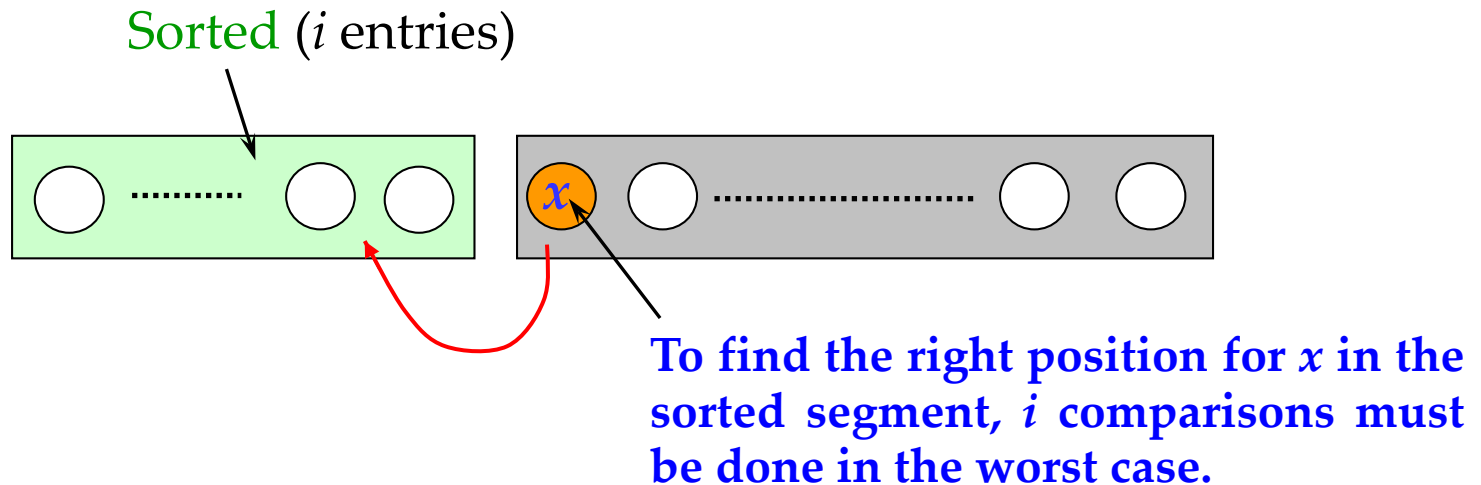
InsertionSort: the Algorithm

- Input: E (array), $n \geq 0$ (size of E)
- Output: E , ordered nondecreasingly 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

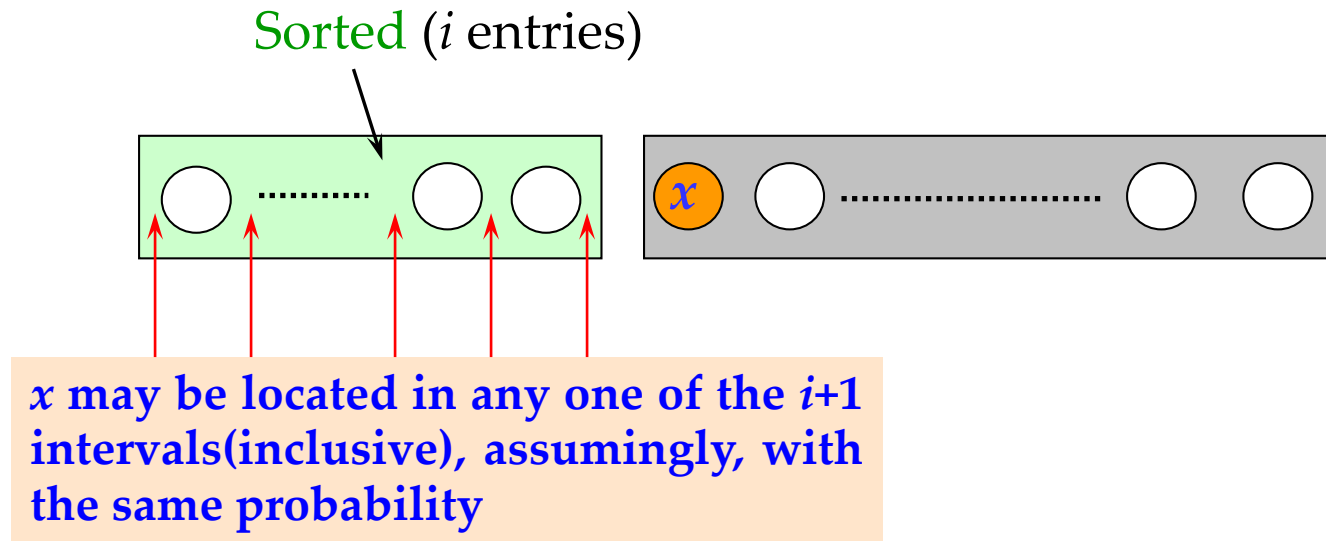


- At the beginning, there are $n-1$ entries in the unsorted segment, so:

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

The input for which the upper bound is reached does exist, so:
 $W(n) \in \Theta(n^2)$

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 $(i+1)$ th interval (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 the leftmost interval

- For all $n-1$ insertions:

$$\begin{aligned} A(n) &= \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right) \\ &= \frac{n^2}{4} + \frac{3n}{4} - 1 - (\ln n + \gamma + \epsilon(n) - 1) \\ &\in \Theta(n^2) \end{aligned}$$

Inversion and Sorting

- An unsorted sequence E :
 - $\{x_1, x_2, x_3, \dots, x_{n-1}, x_n\} = \{1, 2, 3, \dots, n-1, n\}$
- $\langle x_i, x_j \rangle$ 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 $(n, n-1, \dots, 3, 2, 1)$
- The worst-case behavior of any sorting algorithm that remove at most one inversion per key comparison must in $\Omega(n^2)$



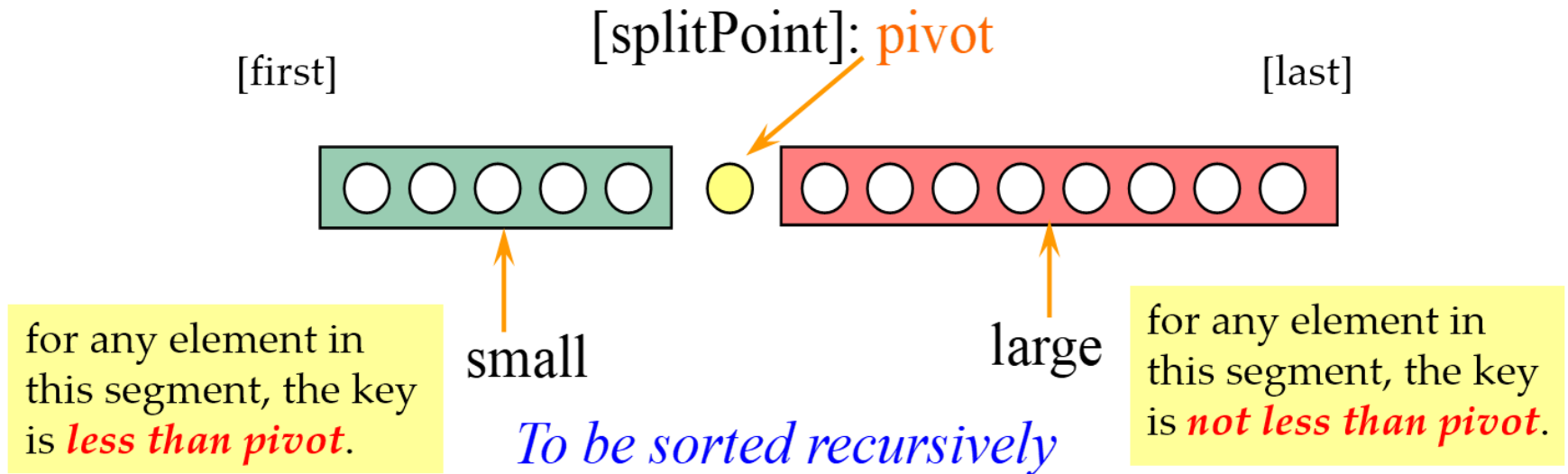
Eliminating Inversions: Average Case

- Computing the average number of inversions in inputs of size n ($n > 1$):
 - Transpose: $x_1, x_2, x_3, \dots, x_{n-1}, x_n$
 $x_n, x_{n-1}, \dots, x_3, x_2, x_1$
 - For any i, j , ($1 \leq j \leq i \leq n$), 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

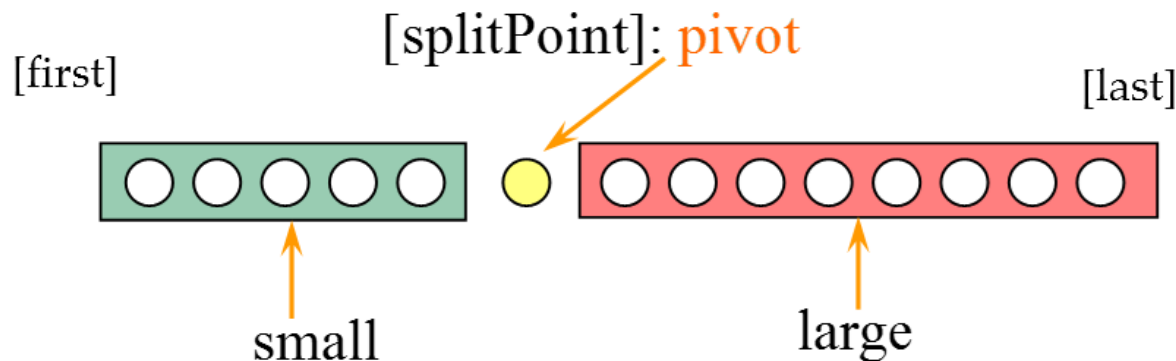
- Divide the array to be sorted into two parts: “small” and “large”, which will be sorted recursively.



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 and indexes $first$, and $last$, such that elements $E[i]$ are defined for $first \leq i \leq last$.
- Output: $E[first], \dots, E[last]$ is a sorted rearrangement of the same elements.

- The procedure:

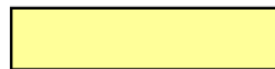
```
void quickSort(Element[ ] E, int first, int last)
    if (first < last)
        Element pivotElement = E[first];
        Key pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first, last);
        E[splitPoint] = pivotElement;
        quickSort(E, first, splitPoint-1);
     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

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

Partition: the Strategy



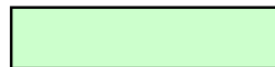
Expanding Directions



“Small” segment



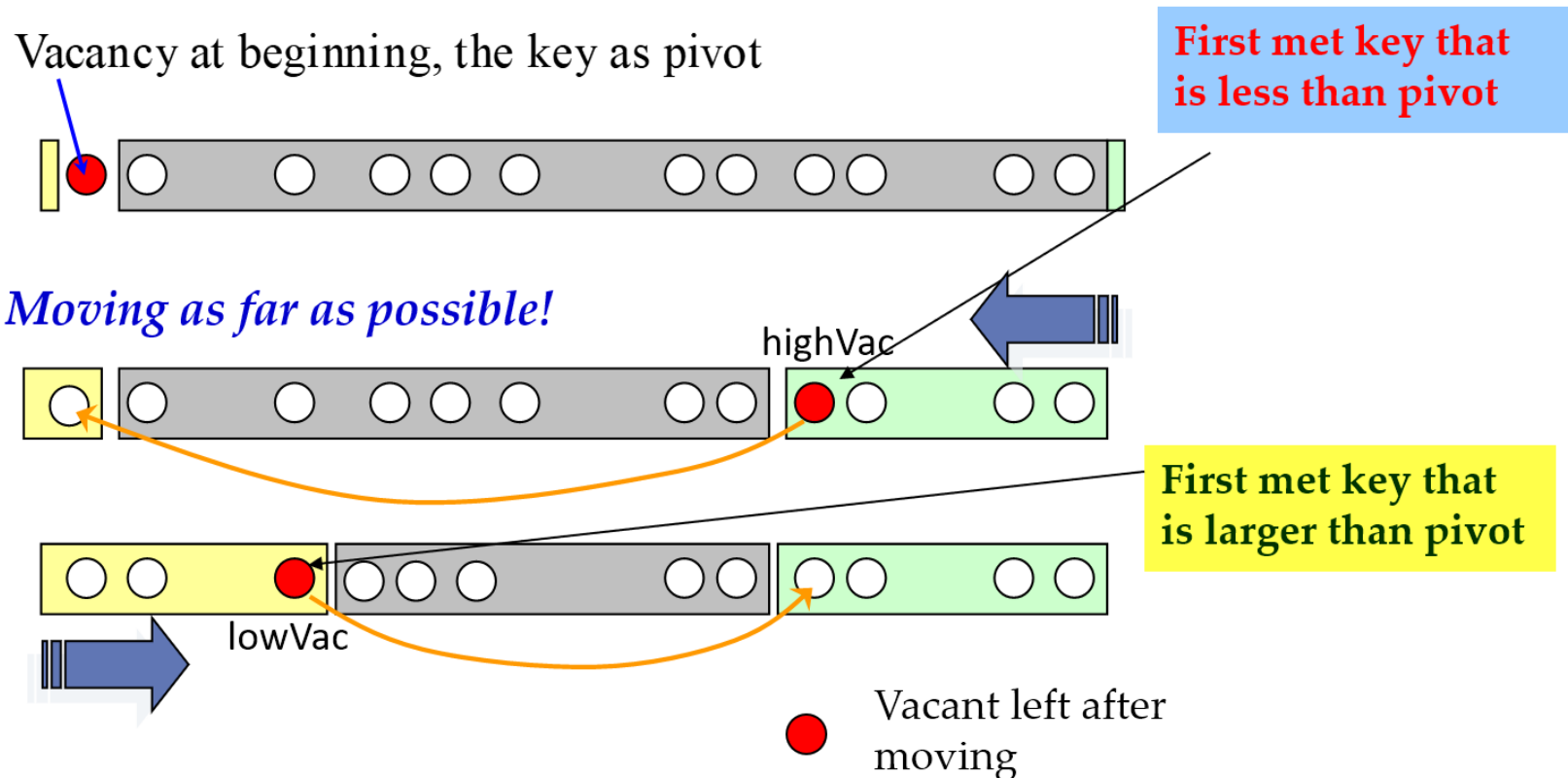
Unexamined segment



“Large” segment

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 $first+1 \leq i \leq last$ and $E[first]$ is vacant. It is assumed that $first < last$.
- Output: Returning $splitPoint$, the elements originally in $first+1, \dots, last$ are rearranged into two subranges, such that
 - the keys of $E[first], \dots, E[splitPoint-1]$ are less than pivot, and
 - the keys of $E[splitPoint+1], \dots, E[last]$ are not less than pivot, and
 - $first \leq splitPoint \leq last$, and $E[splitPoint]$ is vacant.

Partition: the Procedure

```
int partition(Element [ ] E, Key pivot, int first, int last)
    int low, high;
1. low=first; high=last;
2. while (low<high)
3.   int highVac =
      extendLargeRegion(E,pivot,low,high);
4.   int lowVac =
      extendSmallRegion(E,pivot,low+1,highVac);
5.   low=lowVac; high=highVac-1;
6. return low; //This is the splitPoint
```

highVac has been
filled now

Extending Regions

- Specification for

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

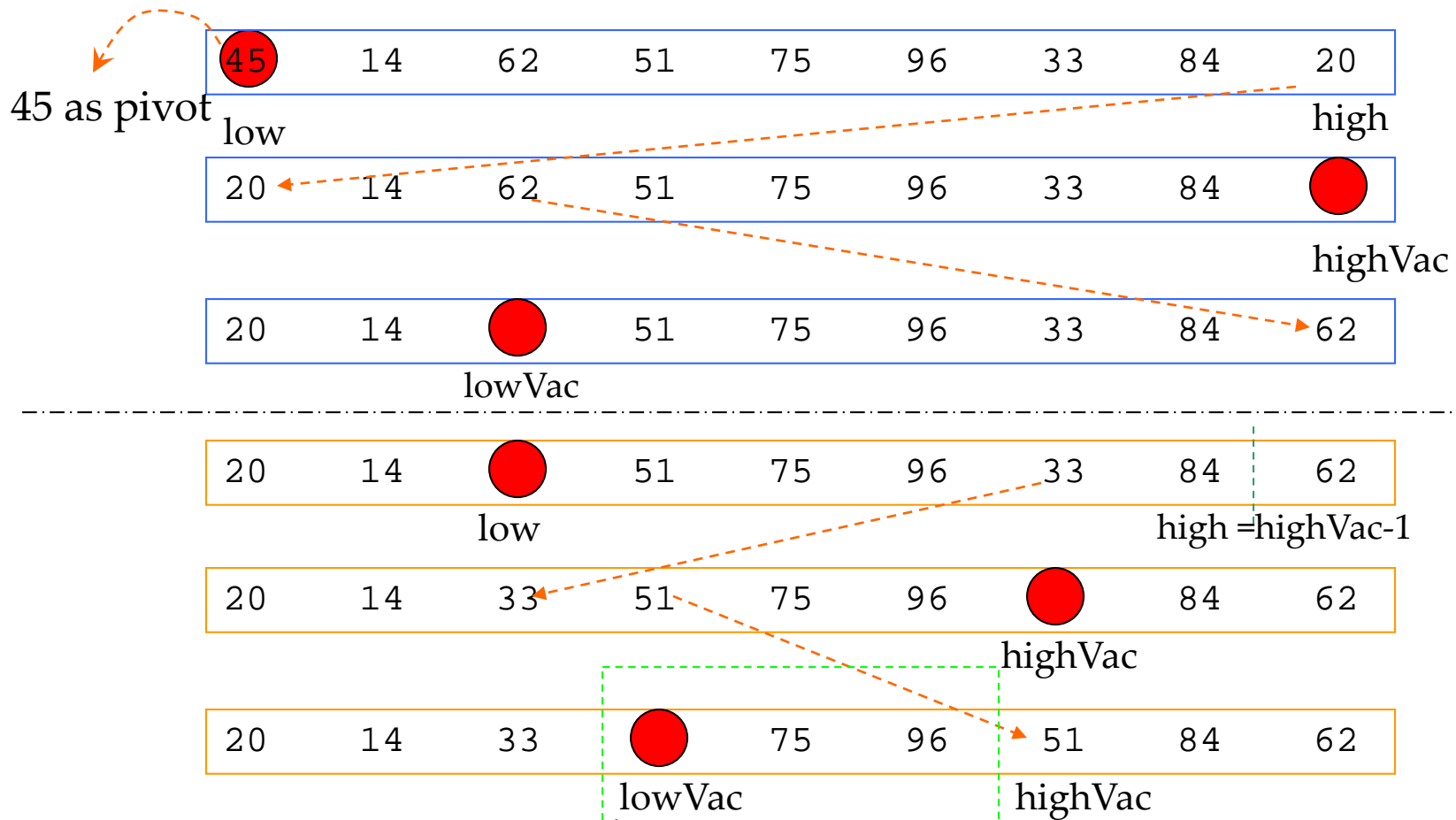
- Precondition:

- lowVac < high

- Postcondition:

- If there are elements in $E[\text{lowVac}+1], \dots, E[\text{high}]$ whose key is less than pivot, then the rightmost of them is moved to $E[\text{lowVac}]$, and its original index is returned.
- If there is no such element, *lowVac* is returned.

An Example



Worst Case: a Paradox

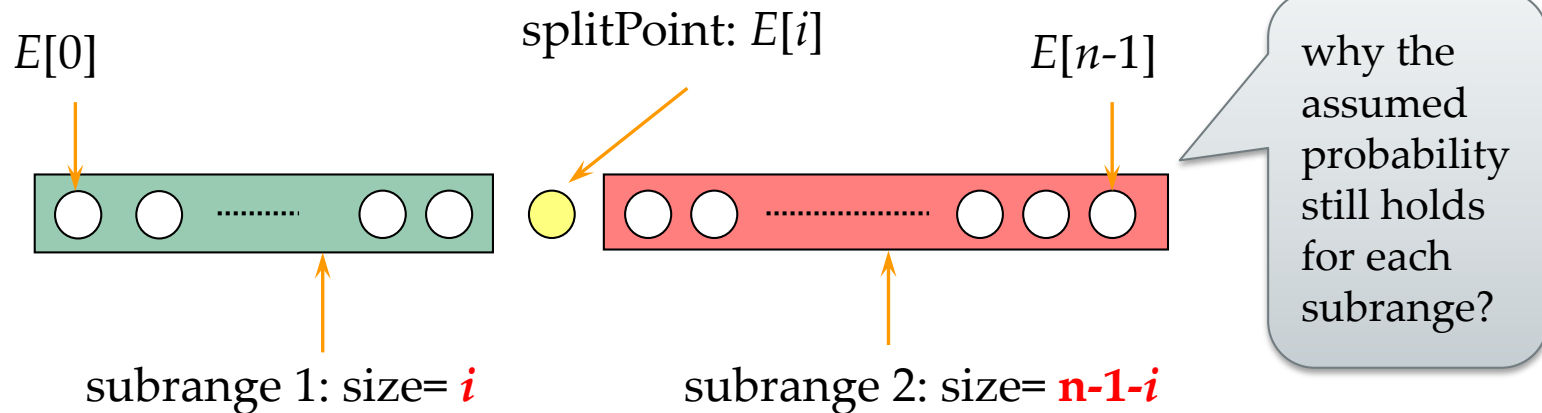
- For a range of k positions, $k-1$ keys are compared with the pivot(one is vacant).
 - If the pivot is the smallest, than the “large” segment has all the remaining $k-1$ elements, and the “small” segment is empty.
 - If the elements in the array to be sorted has already in ascending order(the *Goal*), 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 $[0, \dots, i-1]$ and $[i+1, \dots, n-1]$

The Recurrence Equation



with $i \in \{0, 1, 2, \dots, n-1\}$, each value with the probability $1/n$

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

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

and $A(1)=A(0)=0$

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

Simplified Recurrence Equation

- **Note:** $\sum_{i=0}^{n-1} A(i) = \sum_{i=0}^{n-1} A[(n-1)-i]$ *and* $A(0) = 0$
- **So:** $A(n) = (n-1) + \frac{2}{n} \sum_{i=1}^{n-1} A(i)$ *for* $n \geq 1$
- **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*, case 2:
$$Q(n) \in \Theta(n \log n)$$

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

Inductive Proof:

$A(n) \in O(n \ln n)$

- Theorem: $A(n) \leq cn \ln n$ 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 $1 \leq i < n$

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

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

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

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

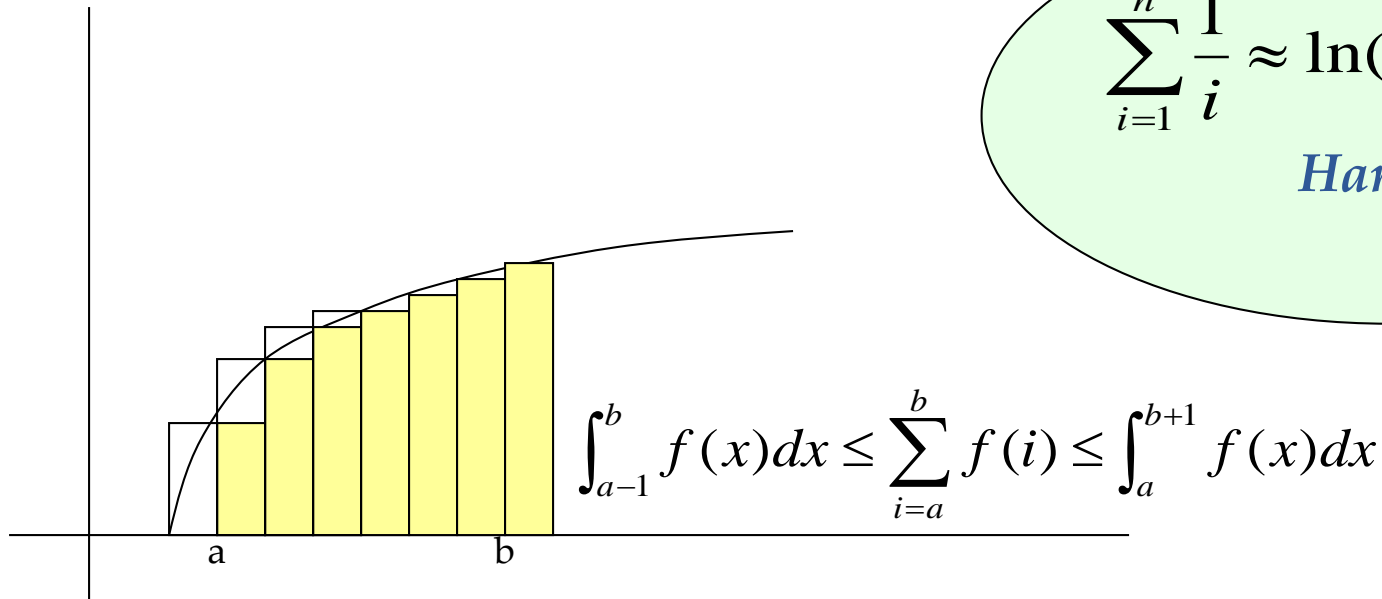


For Your Reference

$$\begin{aligned}\int_1^n x^k \ln x dx &= \left(\frac{x^{k+1} \ln x}{k+1} - \frac{x^{k+1}}{(k+1)^2} \right) \Big|_1^n \\ &= \frac{n^{k+1} \ln n}{k+1} - \frac{n^{k+1}}{(k+1)^2} + \frac{1}{(k+1)^2}\end{aligned}$$

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

Harmonic Series



Inductive Proof:

$A(n) \in \Omega(n \ln n)$

- Theorem: $A(n) > cn \ln n$ for some c , with large n
- Inductive reasoning:

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

Inductive
assumption

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

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

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

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 $i=1,2,\dots,n-2$


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

$$\text{So, } nA(n) = (n+1)A(n-1) + 2(n-1)$$



Solve the Equation

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

 Let it be $B(n)$

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

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

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

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**
 - $O(n \log n)$
- **k-Sorted**
 - $O(n \log k)$

Thank you!

Q & A

Yu Huang

<http://cs.nju.edu.cn/yuhuang>

