Sorting

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

The Sorting Problem

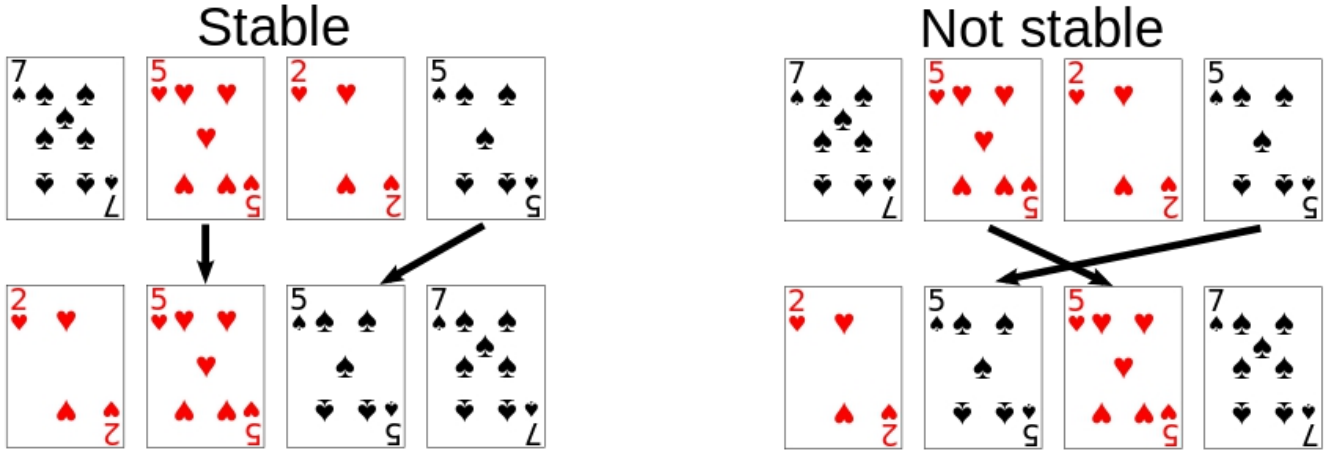
- Sort n numbers into ascending order.
- We can actually sort a collection of any type of data, so long as a total order is defined for that type of data.
  - **Example:** strings, dates, …
- That is, for data items a and b, we can determine: $a < b$, $b < a$, or neither, where "<" is a binary relation.
  - **Example:** in C++, to use sort in STL for sorting, you should define bool compare(DataType item1, DataType item2)
- We can also sort partially ordered items (more on this later).

Sorting algorithms till now

- **Insertion Sort:** gradually increase size of sorted part.
  - $O(n^2)$ time, $O(1)$ space.
- **Merge Sort:** textbook example of divide-and-conquer.
  - $O(n \log n)$ time, $O(n)$ space.
- **Heap Sort:** leverage the heap data structure.
  - $O(n \log n)$ time, $O(1)$ space.

Characteristics of sorting algorithms

- **In-place:** a sorting algorithm is *in-place* if $O(1)$ extra space is needed beyond input.
- **Stability:** a sorting algorithm is *stable* if numbers with the same value (i.e., items that are "=") appear in the output array in the same order as they do in the input array.

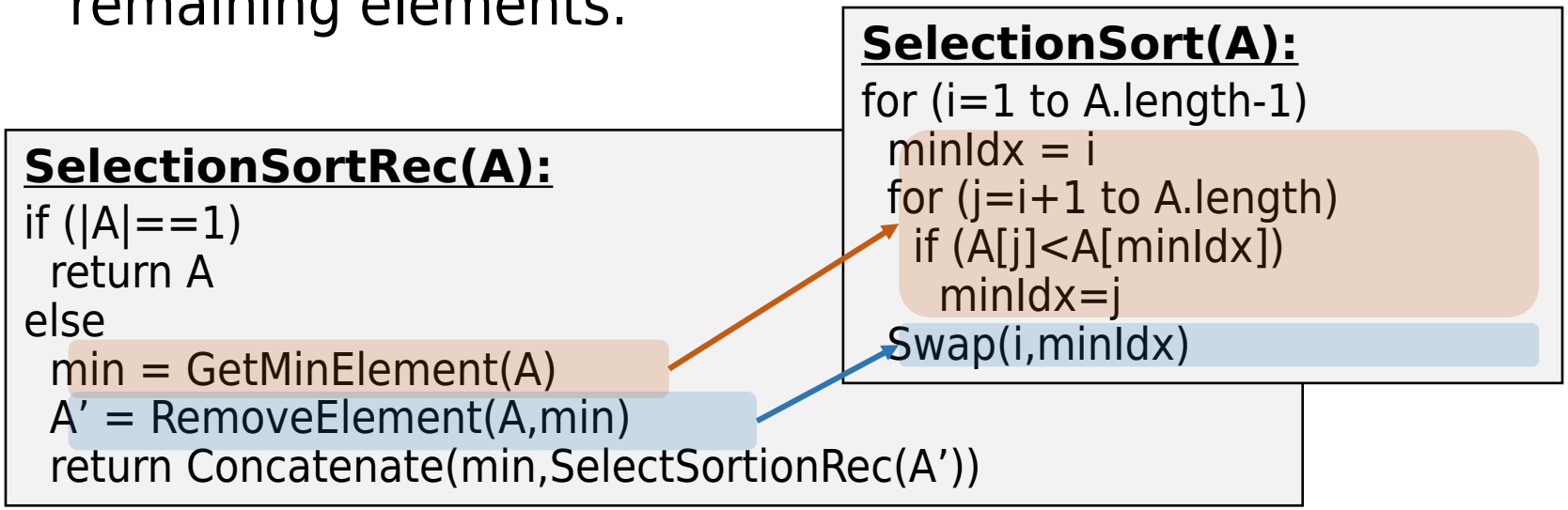


Sorting algorithms till now

- **Insertion Sort:** gradually increase size of sorted part.
  - $O(n^2)$ time, $O(1)$ space.
  - In-place, and stable.
- **Merge Sort:** textbook example of divide-and-conquer.
  - $O(n \log n)$ time, $O(n)$ space.
  - Not in-place (for typical implementation), but stable.
- **Heap Sort:** leverage the heap data structure.
  - $O(n \log n)$ time, $O(1)$ space.
  - In-place, but not stable. (Information about relative ordering of identically valued items was lost during heap creation.)
  - Counterexample for stability: <2a, 2b, 1>

The Selection Sort Algorithm

- **Basic idea:** pick out minimum element from input, then recursively sort remaining elements, and finally concatenate the minimum element with sorted remaining elements.



Analysis of **SelectionSort**

- Why it is correct? (What is the loop invariant?)
  - After the i^{th} iteration, the first i items are sorted, and they are the i smallest elements in the original array.
- Time complexity for sorting n items?
  - $\sum_{i=1}^{n-1}(\Theta(1)+\Theta(n-i))=\Theta(n^2)$
- Space complexity?
  - $O(1)$ extra space, thus in-plac
- Stability?
  - Not stable! Swap operation ca
    mess up relative order.
  - Example: <2a, 2b, 1>

SelectionSort(A):

```
for (i=1 to A.length-1)
  minIdx = i
  for (j=i+1 to A.length)
    if (A[j]<A[minIdx])
      minIdx=j
  Swap(i,minIdx)
```

Before we move on...

SelectionSort(A):

```
for (i=1 to A.length-1)
  minIdx = i
  for (j=i+1 to A.length)
    if (A[j]<A[minIdx])
      minIdx=j
  Swap(i,minIdx)
```

SelectionSortRec(A):

```
if (|A|==1)
  return A
else
  min = GetMinElement(A)
  A' = RemoveElement(A,min)
  return Concatenate(min,SelectSortionRec(A'))
```

SelectionSortRec(A):

```
if (|A|==1)
  return A
else
  max = GetMaxElement(A)
  A' = RemoveElement(A,max)
  return Concatenate(SelectSortionRec(A'),max)
```

What if A is organized as a heap?

This framework leads to the faster HeapSort algorithm.

The Bubble Sort Algorithm

- **Basic idea:** repeatedly step through the array, compare adjacent pairs and swaps them if they are in the wrong order. Thus, larger elements "bubble" to the "top".

```
BubbleSort(A):
for (i=A.length downto 2)
  for (j=1 to i-1)
    if (A[j]>A[j+1])
      Swap(A[j],A[j+1])
```

- Correctness: what is the loop invariant
- Time complexity: $\Theta(n^2)$
- Space complexity: $O(1)$
- Stability: stable.

Improving **BubbleSort**

BubbleSort(A):

```
for (i=A.length downto 2)
  for (j=1 to i-1)
    if (A[j]>A[j+1])
      Swap(A[j],A[j+1])
```

What if in one iteration we never swap data items?

Then $A[1...i]$ are sorted, and we are done! (Why?)

BubbleSortImproved(A):

```
n=A.length
repeat
  swapped=false
  for (j=1 to n-1)
    if (A[j]>A[j+1])
      Swap(A[j],A[j+1])
      swapped=true
  n=n-1
until (swapped==false)
```

When the input is mostly sorted, this variant performs much better. Particularly, when the input is sorted, this variant has $O(n)$ runtime.

Q: Other algorithms that also have this property?

A: such as InsertionSort.

Nonetheless, the worst case performance is still $\Theta(n^2)$. E.g., when input is reversely sorted.

Improving **BubbleSort**

BubbleSortImproved(A):

```
n=A.length
repeat
  swapped=false
  for (j=1 to n-1)
    if (A[j]>A[j+1])
      Swap(A[j],A[j+1])
      swapped=true
  n=n-1
until (swapped==false)
```

BubbleSortImprovedAgain(A):

```
n=A.length
repeat
  lastSwapIdx=-1
  for (j=1 to n-1)
    if (A[j]>A[j+1])
      Swap(A[j],A[j+1])
      lastSwapIdx=j+1
  n=lastSwapIdx-1
until (n<=1)
```

Can we do better?

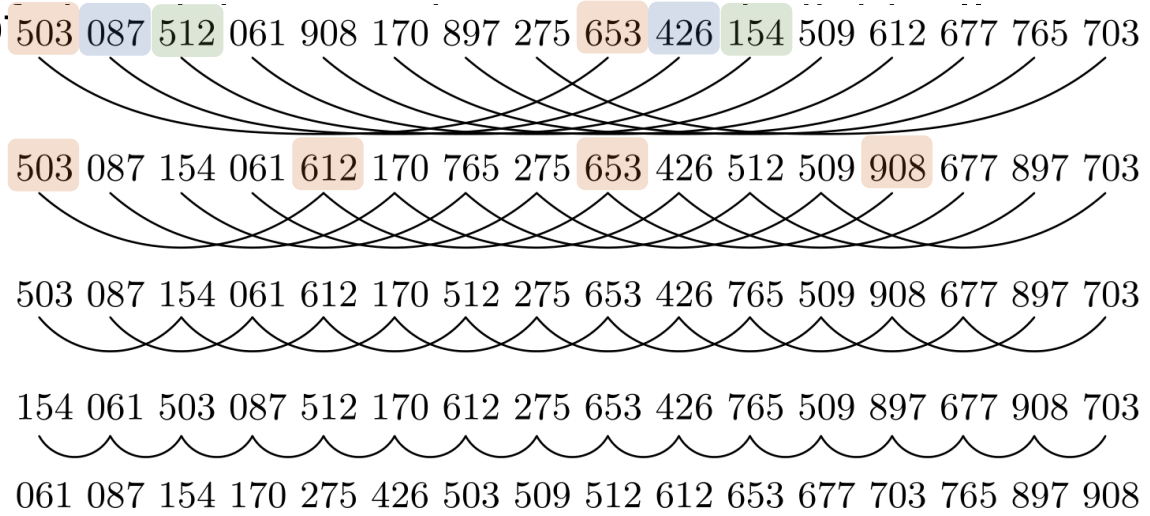
We can be more aggressive when reducing n after each iteration: in $A[1...n]$, items after the last swap are all in correct sorted position. (Why?)

$n = 5$

3	2	1	8	9	12	15	
3	2	1	8	9	12	15	Swap!
2	3	1	8	9	12	15	Swap!
2	1	3	8	9	12	15	No swap!
2	1	3	8	9	12	15	No swap!
2	1	3	8	9	12	15	$n = 2$

*Shell's method for sorting

- Let's first see an example of **ShellSort**: sort 16 integers.
- **[Pass 1]** Group elements of distance 8 together, end up with eight groups each of size two. Sort these groups individually.
- **[Pass 2]** Group elements of distance 4 together, end up with four groups each of size four. Sort these groups individually.
- **[Pass 3]** Group elements of distance 2 together, end up with two groups each o
- **[Pass 4]** Group elements of distance 1, this is just an ordinary sort on all elements.



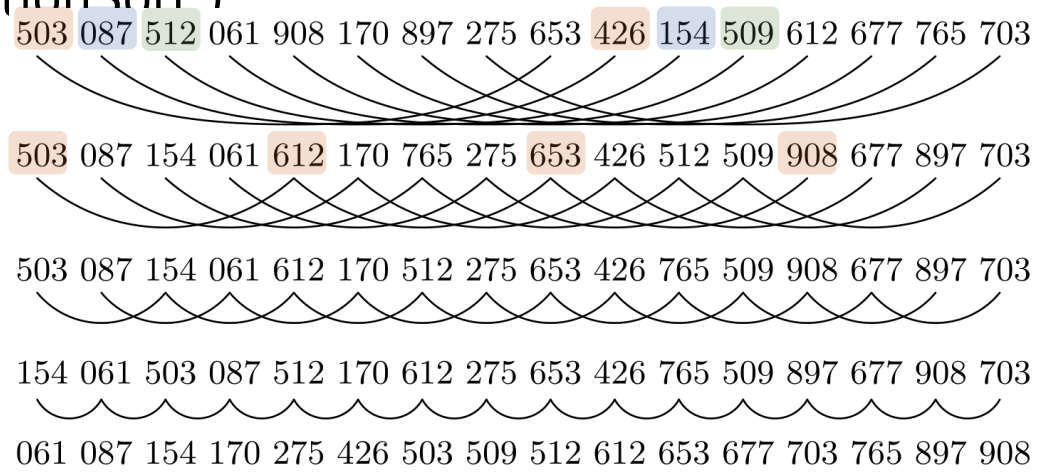
General framework of **ShellSort**

- To sort n items, define a set of decreasing distances $\{d_1, d_2, \cdots, d_k\}$ with $d_1 < n$ and $d_k = 1$.
- ShellSort then go through k passes, for the i^{th} pass:
  - Divide items into d_i groups each of size about n/d_i, and the j^{th} group contains items with index $j,\ \ j + d_i,\ \ j + 2d_i,\ \ j + 3d_i, \cdots$
  - For each of the d_i groups, sort the items in that group. (Usually uses InsertionSort.)

Why ShellSort is correct?

The last pass always sort all items since $d_k = 1$.

But then why bother earlier passes?!



Motivation of **ShellSort**

- In a sequence of items $\langle a_1, a_2, \cdots, a_n \rangle$, if $i < j$ and $a_i > a_j$, then the pair (a_i, a_j) is call an **inversion**.
- The process of sorting is to correct all inversions!

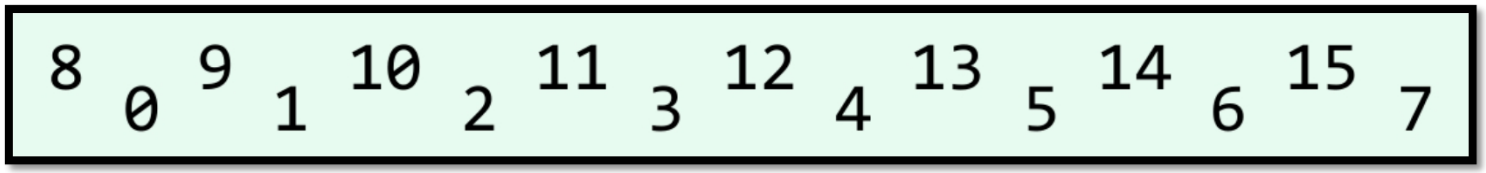
- Earlier passes in ShellSort reduce number of inversions, making the sequence “closer” to being sorted.
- InsertionSort performs better (i.e., faster) as the input sequence becomes “closer” to being sorted.

Ideal versus Reality

- Unfortunately, ShellSort is not that fast, at least when using Shell's original distances…
- Upper bound on the runtime of ShellSort:
  - Assume we have n items where n is some power of two.
  - The distances are $n/2, n/4, \cdots, 1$.
  - For the ith pass, we run $n/2^i$ instances of InsertionSort, each having to sort 2^i items.
  - So the total runtime is $\sum_{i=1}^{(\lg n)-1} \left(n/2^i \cdot O(2^{2i}) \right) = O(n^2)$
- Will ShellSort actually perform so poor?

ShellSort can be slow!

- When using Shell's original distances, the runtime of ShellSort can be $\Theta(n^2)$ for certain input sequences.
- **Example:** input is $[n]$, where $[n/2]$ are in even positions, and $[n] \setminus [n/2]$ are in odd positions.



- Then, before the last pass, no pair (a_i, a_j) where i and j are out of order is ever compared!
- In the last pass, $\Theta(n^2)$ work has to be done!

Choice of distances matters, a lot!

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A168604	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Pratt, 1971,[1] Knuth, 1973[3]
A036569	$\prod_I a_q$, where $a_q = \min\left\{n \in \mathbb{N} : n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p : 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\right\}$, $I = \left\{0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k\right\}$, $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8 \ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1982[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{\left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1\right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11}\left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right) \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]

Divide-and-Conquer:

A unified view for many sorting algs.

- Divide the input into size 1 and size $n-1$.
  - InsertionSort, easy to divide, combine needs efforts.
  - SelectionSort, divide needs efforts, easy to combine.
- Divide the input into two parts each of same size.
  - MergeSort, easy to divide, combine needs efforts.
- Divide the input into two parts of approximately same size.
  - QuickSort, divide needs efforts, easy to combine.

Divide problem into subproblems.

Conquer subproblems recursively.

Combine solutions of

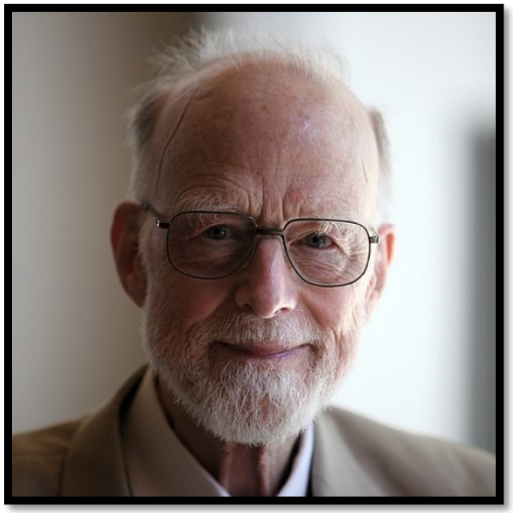
The **QuickSort** Algorithm

```
QuickSortAbs(A):
x = GetPivot(A)
<B,C> = Partition(A,x)
QuickSortAbs(B)
QuickSortAbs(C)
return Concatenate(B,x,C)
```

- **Input:** array A of n iter
- **Algorithms:**
  - Choose one item x in A as the pivot.
  - Use the pivot to partition the input into B and C, so that items in B are $\leq x$, and items in C are $> x$.
  - Recursively sort B and C.
  - Output $\langle B, x, C \rangle$.

C. A. R. Hoare

Recipient of the Turing Award in 1980 for "fundamental contributions to the definition and design of programming languages".



Choosing the pivot

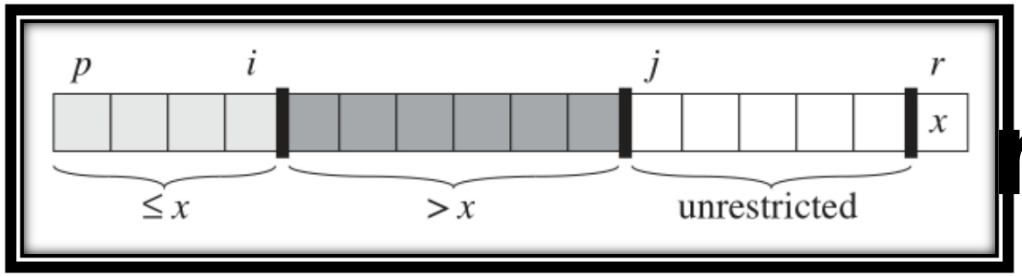
- Ideally the pivot should partition the input into two parts of roughly same size (we'll see why later).
- $A[1]$, $A[n]$, $A[\,n/2\,]$, median of $\{A[1], A[n], A[\,n/2\,]\}$?
- For every simple *deterministic* method of choosing pivot,
  we can construct corresponding "*bad input*".
- For now just use the last item as the pivot.

The **Partition** Procedure

- Allocate array B of size n.
- Sequentially go through $A[1...(n-1)]$, put small items at the left side of B, and large items at the right side of B.
- Finally put the pivot in the (only) remaining position
- $\Theta(n)$ time, $\Theta(n)$ space, unsta
- Can we do better, and how?

Partition(A):

```
x = A[n], l = 1, r = n
for (i=1 to n-1)
  if (A[i]<=x)
    B[l] = A[i]
    l++
  else
    B[r] = A[i]
    r--
B[l]=x
return <B,l>
```

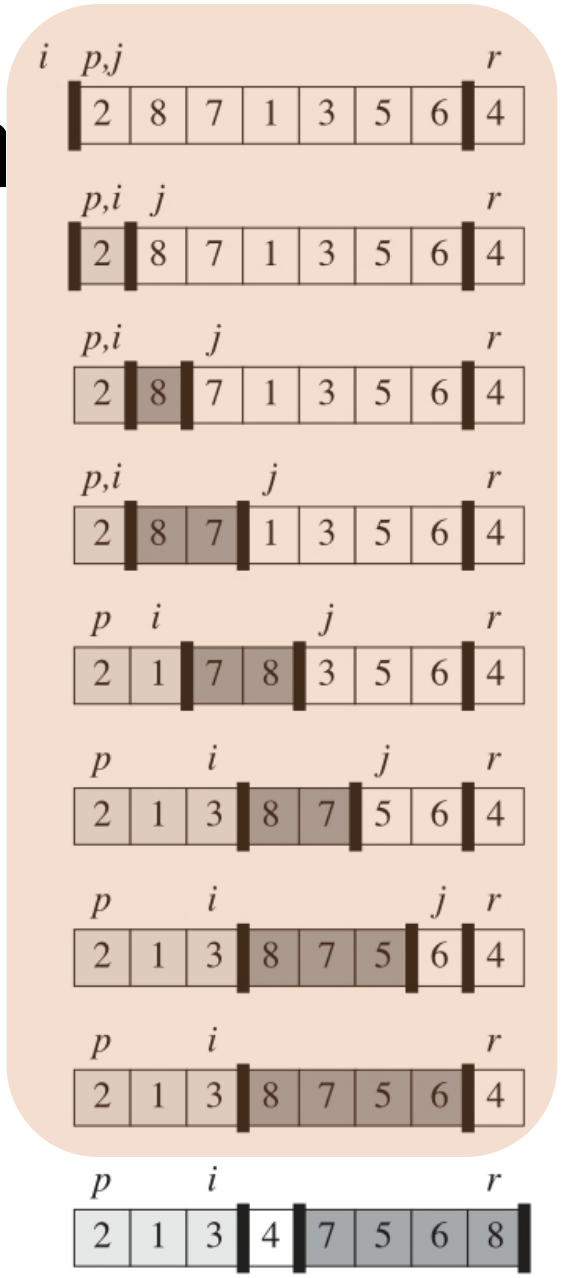


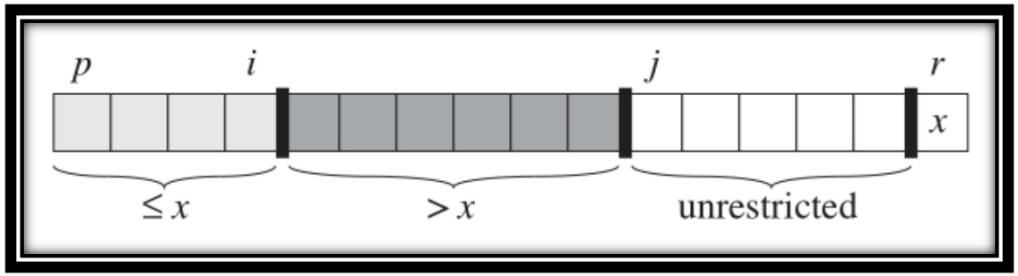
n

- **Basic Idea:** sequentially go through A, use swap operations to move small items to the left part of A; thus the right part of A naturally contains large

InplacePartition(A, p, r):

```
x = A[r], i=p-1
for (j=p to r-1)
  if (A[j]<=x)
    i=i+1
    Swap(A[i],A[j])
Swap(A[i+1],A[r])
return i+1
```





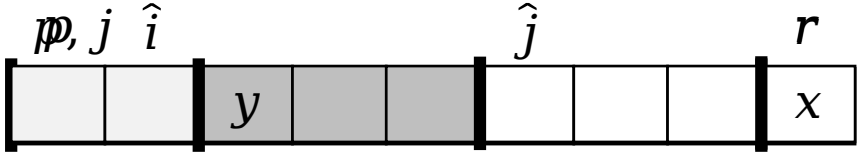
InplacePartition(A, p, r):

```
x = A[r], i=p-1
for (j=p to r-1)
  if (A[j]<=x)
    i=i+1
    Swap(A[i],A[j])
Swap(A[i+1],A[r])
return i+1
```

Claim: at the beginning of any iteration, for any index k:

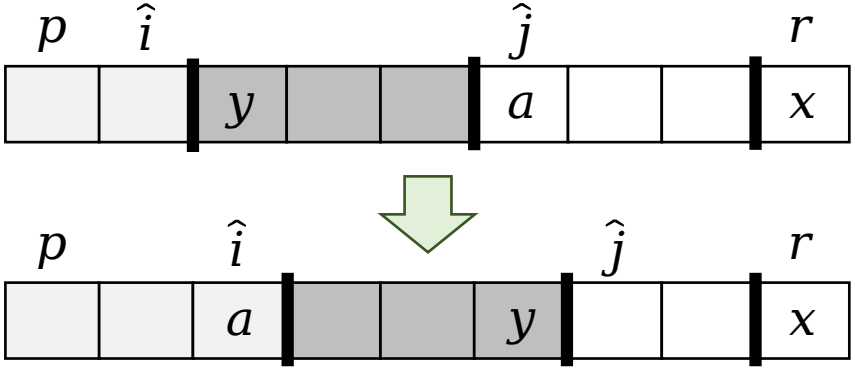
1. If $k \in [p, i]$, then $A[k] \leq x$;
2. If $k \in [i+1, j-1]$, then $A[k] > x$;
3. If $k = r$, then $A[k] = x$.

Proof: we use induction.

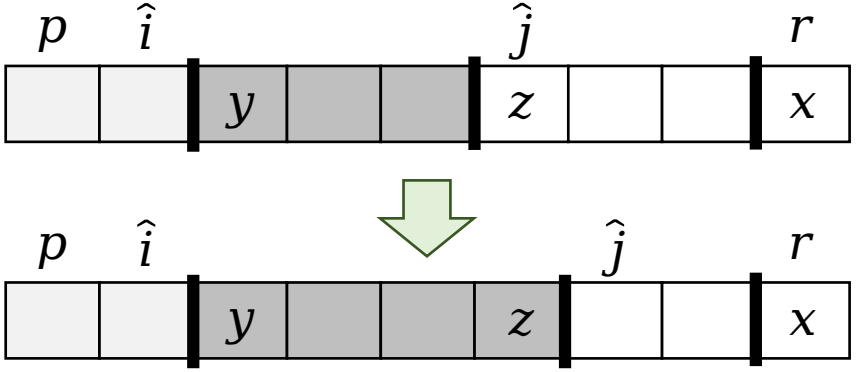


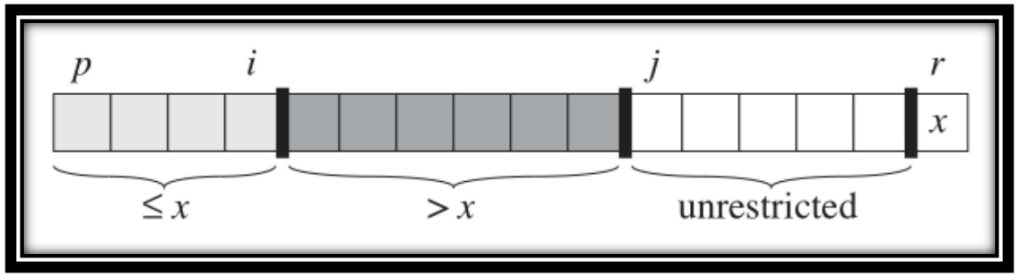
[*Basis*] Trivially holds.

[*Inductive step*] Assume at the beginning of some iteration we have $i = \hat{i}$ and $j = \hat{j}$, and the stated properties hold. In this iteration:



or





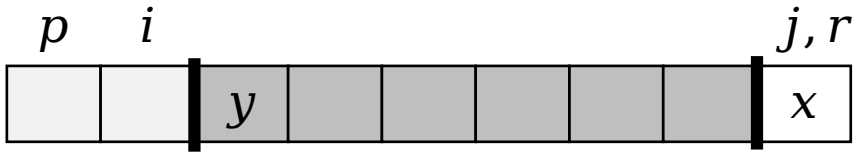
InplacePartition(A, p, r):

```
x = A[r], i=p-1
for (j=p to r-1)
  if (A[j]<=x)
    i=i+1
    Swap(A[i],A[j])
Swap(A[i+1],A[r])
return i+1
```

Claim: at the beginning of any iteration, for any index k:

1. If $k \in [p, i]$, then $A[k] \leq x$;
2. If $k \in [i+1, j-1]$, then $A[k] > x$;
3. If $k = r$, then $A[k] = x$.

Eventually, when $j = r$:



Swap $A[i+1]$ and $A[r]$:

During execution, we only swap items, no addition/deletion.

So **InplacePartition** correctly partitions the input array.

The **QuickSort** Algorithm

InplacePartition(A, p, r):

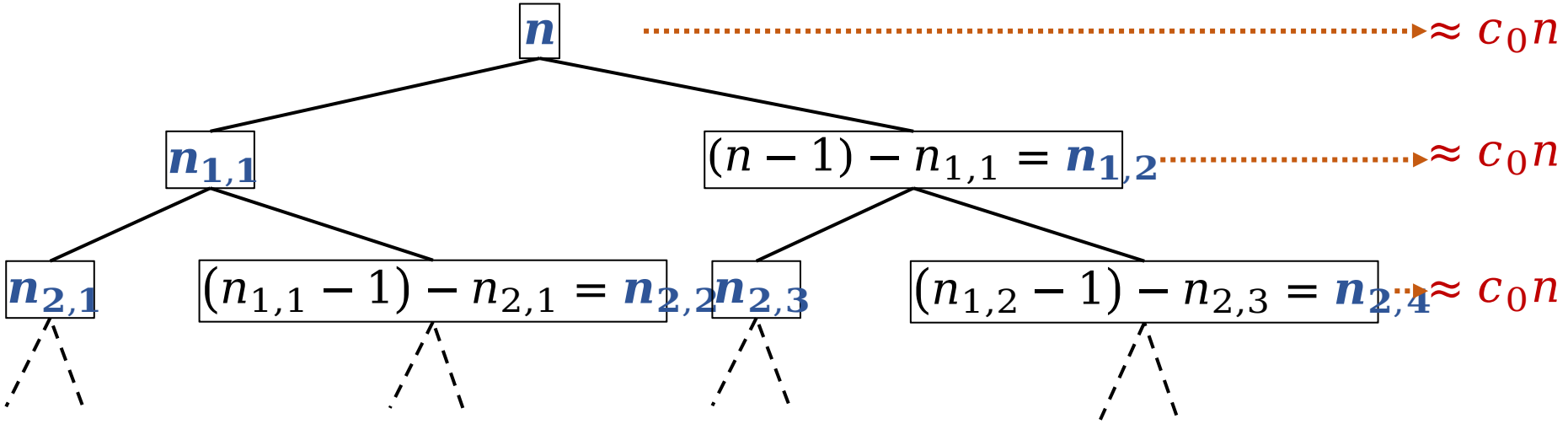
```
x = A[r], i=p-1
for (j=p to r-1)
  if (A[j]<=x)
    i=i+1
    Swap(A[i],A[j])
Swap(A[i+1],A[r])
return i+1
```

QuickSort(A, p, r):

```
if (p<r)
  q = InplacePartition(A,p,r)
  QuickSort(A,p,q-1)
  QuickSort(A,q+1,r)
```

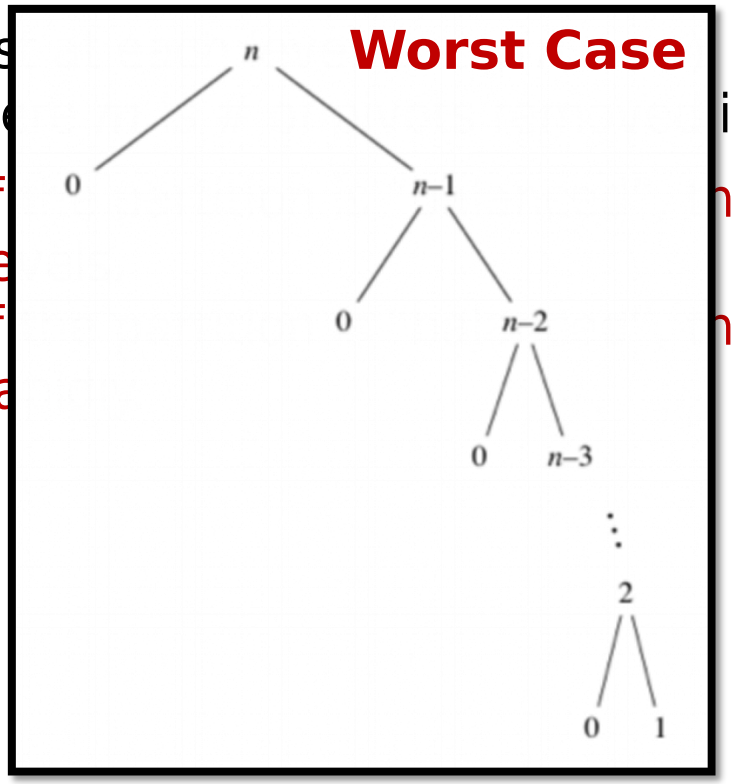
Performance of InplacePartition:
$\Theta(|r-p|)$ time (i.e., linear time); $O(1)$ space; unstable.

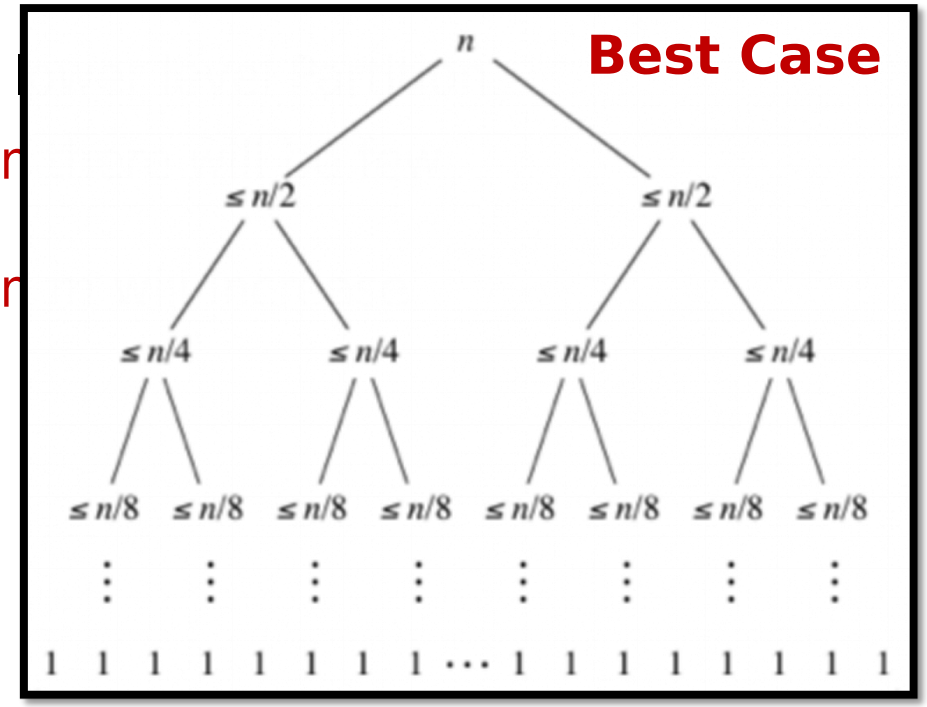
Performance of QuickSort?



Cos
whe
in

- If
  le
  her
- If
  ra
  her





Performance of **QuickSort**

- Recurrence for the worse-case runtime of QuickSort:

$$T(n) = \max_{0 \le q \le n-1} \left(T(q) + T(n-q-1)\right) + c_0 n$$

- Guess $T(n) \le cn^2$, and we now verify:

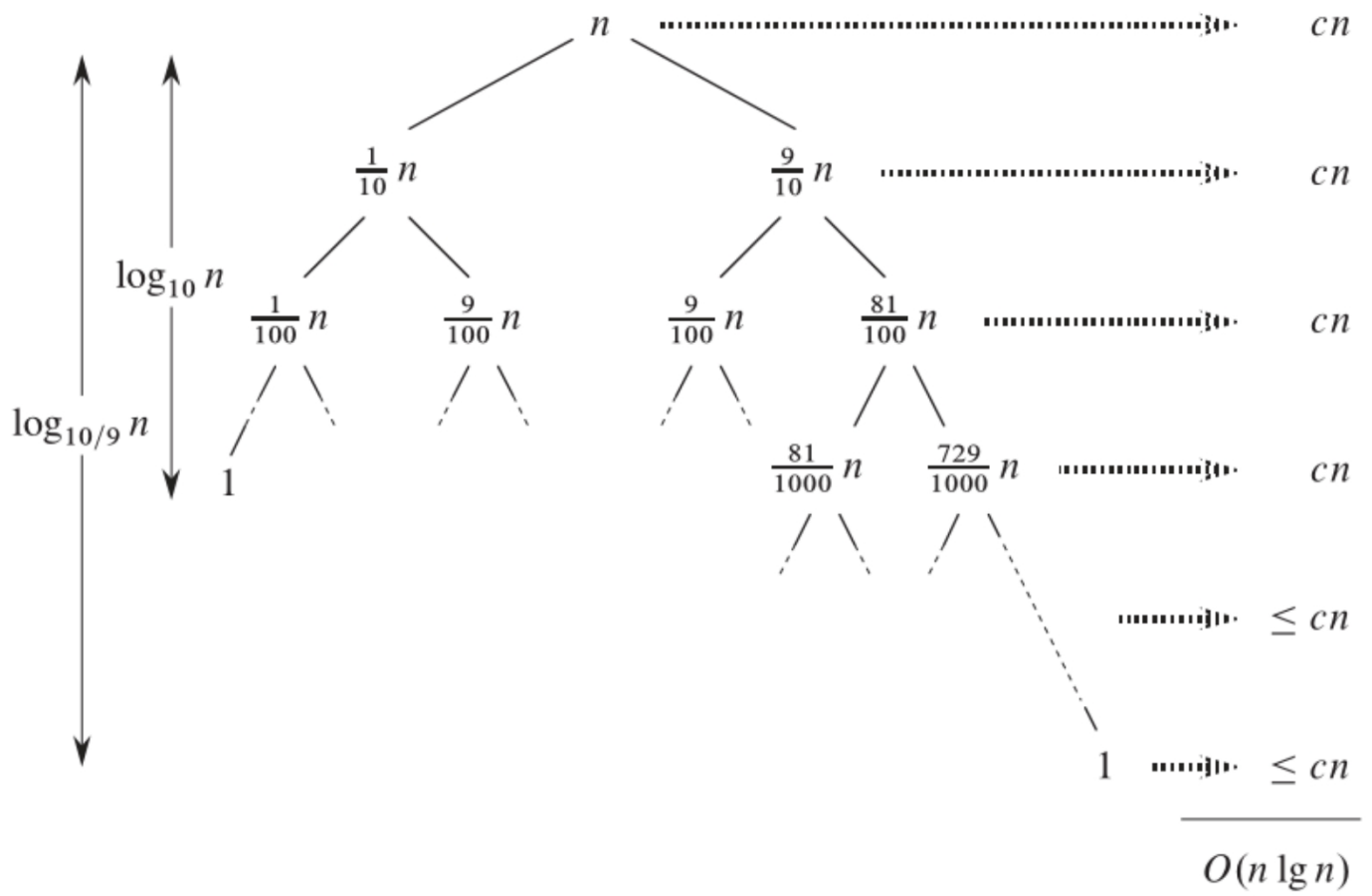
$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} \left(cq^2 + c(n-q-1)^2\right) + c_0 n \\ &= c \cdot \max_{0 \le q \le n-1} \left(q^2 + (n-q-1)^2\right) + c_0 n \\ &\le c(n-1)^2 + c_0 n = cn^2 - c(2n-1) + c_0 n \\ &\le cn^2 \end{aligned}$$

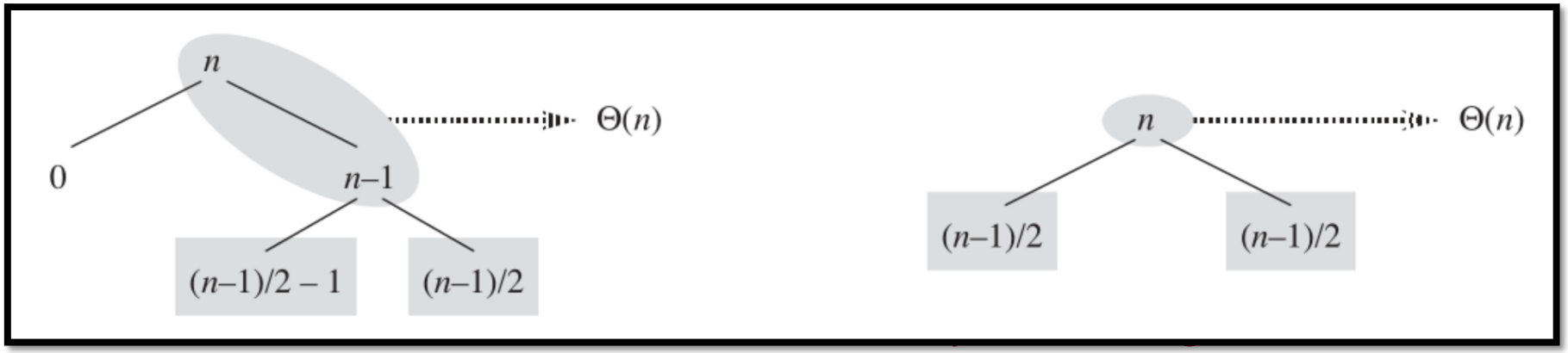
when $q = 0$ or $q = n-1$

QuickSort(A, p, r):

```
if (p<r)
  q = InplacePartition(A,p,r)
  QuickSort(A,p,q-1)
  QuickSort(A,q+1,r)
```

Performance of **QuickSort**





- **Average-case analysis:** given a set of n distinct numbers, assume each of the $n!$ permutations is *equally* likely to be the input, what is the *expected* runtime of QuickSort?
- Each call to Partition is quite likely to be reasonably balanced (i.e., each split is of constant proportionality).
- So there cannot have too many "bad" Partition between two "good" Partition.
- The cost of "bad" Partition can be absorbed by recent "good" Partition, without affecting time complexity asymptotically.
- The average runtime of QuickSort is $O(n \log n)$.

Randomized **QuickSort**

RndQuickSort(A, p, r):

```
if (p<r)
  i = Random(p,r)
  Swap(A[r],A[i])
  q = InplacePartition(A,p,r)
  RndQuickSort(A,p,q-1)
  RndQuickSort(A,q+1,r)
```

rtant for the performance,

, median of three, …?

echanism could fail!

sary" that knows the

algorithm.)

- Choose pivot (uniformly) **at random**!
- Since the choice is randomly made, there is a good chance (constant probability) that we choose a "good" pivot.
- The above claim holds even if the input is given by an "adversary" that knows the algorithm (but not the random bits the algorithm uses).

Performance of **RndQuickSort**

RndQuickSort(A, p, r):

```
if (p<r)
  i = Random(p,r)
  Swap(A[r],A[i])
  q = InplacePartition(A,p,r)
  RndQuickSort(A,p,q-1)
  RndQuickSort(A,q+1,r)
```

InplacePartition(A, p, r):

```
x = A[r], i=p-1
for (j=p to r-1)
  if (A[j]<=x)
    i=i+1
    Swap(A[i],A[j])
Swap(A[i+1],A[r])
return i+1
```

The expected runtime is $O(n \log n)$, for ***any*** input distribution.

In an execution of RndQuickSort, the cost is $O(n) + O(total\ number\ of\ comparisons)$.

- Recursive RndQuickSort.

Each node can be pivot at most once!

For each InplacePartition, cost is $O(\#\ of\ comparisons)$.

Cost of RndQuickSort is $O(n + X)$, where X is a r.v. denoting the number of comparisons happened in InplacePartition throughout entire execution.

Each of pair of items is compared at most once!

(Items only compare with pivots, and each item can be the pivot at most once.)

Let $X_{ij} = I\{z_i \; is \; ever \; compared \; to \; z_j\}$, where z_i is the item with ran

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\Pr(X_{ij}=1)$$

Let $Z_{ij} = [z_i, z_j]$ where $i \le j$, let $\widehat{z_{ij}}$ be the first item in Z_{ij} that is chosen as a pivot. Then, z_i and z_j are compared iff $\widehat{z_{ij}} = z_i$ or $\widehat{z_{ij}} = z_j$.

(Items from Z_{ij} stay in same split until some pivot is chosen from Z_{ij}.)

$$\Pr(X_{ij}=1) = \Pr(\widehat{z_{ij}} = z_i) + \Pr(\widehat{z_{ij}} = z_j) = \frac{2}{j-i+1}$$

$\mathbb{E}[X] = O(n\log n)$, thus expected runtime of RndQuickSort is $O(n\log n)$.

In fact, runtime of RndQuickSort is $O(n\log n)$ with high probability!

A bit more on Quick Sort

- What if there are many duplicates?
  - Maintain four regions as we go through the array.

  - End up with three regions (“<”, “=”, and “>”), and only recurse into two of them (“<” and “>”). *Wow! Worst-cast becomes best-case!*
- Stop recursion once the array is too small.
  - Recursion has overhead, QuickSort is slow on small arrays.
- Multiple pivots?
  - Early studies do not give promising results…
  - Dual-Pivot variant proposed by Yaroslavskiy in 2009 seems faster.
  - This variant is used in Java for sorting. (Since Java 7.)
  - “Average Case Analysis of Java 7's Dual Pivot Quicksort”. (Best Paper of ESA 2012)

Summary on Quick Sort

- **A widely-used efficient sorting algorithm.**

- Easy to understand! (divide-and-conquer…)
- Moderately hard to implement correctly. (partition…)
- Harder to analyze. (randomization…)
- Challenging to optimize. (theory and practice…)

Reading

- [CLRS] Ch.7, Appendix C (C.2-C.4) on probability theory
- *On **ShellSort**:
- [Weiss] Ch.7 (7.4)
- [Deng] Ch.12 (12.3)
- [TAOCP] Ch.5 (5.2.1 in vol.3)