Analysis of Algorithms

COMP.4040, Summer 2019

Chapter 5&7: Quick Sort & Probabilistic Analysis and Randomized Algorithms

By: Sirong Lin, PhD

UMASS LOWELL
Learning with Purpose

University of Massachusetts Lowell

Outline

Quick Sort

Probabilistic analysis

Randomized algorithms

Indicator random variables

Quick Sort analysis (Cont’d)

Homework 4

Due Date: June 10 (M)

BEFORE the class begins

Quick Sort

Quick Sort

developed by Tony Hoare in 1959

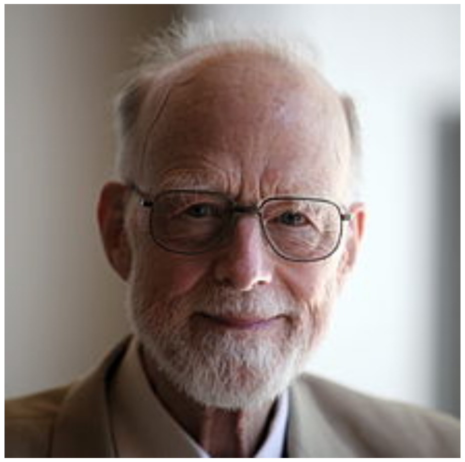
efficient algorithm in practice (with tuning)

- 2~3 times faster than merge sort and heap sort

sorts “in place”

- merge sort used extra space

uses Divide and Conquer



Quick Sort — Divide and Conquer

Three-step to sort subarray A[p..r]

- **Divide**: by partitioning the input array into 2 subarrays around pivot A[q], such that
  - each element in A[$p..q$-1] ≤ A[q]
  - A[q] ≤ each element in A[q+1..r]
  - compute the index q
- **Conquer**: reclusively sort 2 subarrays
- **Combine**: trivial (two subarrays are already sorted)

Quick Sort — Algorithm

Write the recursive quick sort algorithm by yourself

```
QUICKSORT($A, p, r$)
1  if $p < r$
2      $q$ = PARTITION($A, p, r$)
3      QUICKSORT($A, p, q - 1$)
4      QUICKSORT($A, q + 1, r$)
```

Initial call: QUICKSORT(A, 1, n)

Quick Sort — Divide and Conquer (Cont’d)

- Key step — **partition** step (PARTITION)
  - does all the work
  - returns the index *q* that marks the position separating the subarrays
- Quick sort: recursively partitioning
  - partition, quick sort, quick sort

Quick Sort — Partitioning

array is partitioned into four regions (could be empty)

- each element in region1 ≤ pivot
- each element in region2 > pivot
- pivot
- region4: not examined

Quick Sort — Partitioning (Cont’d)

```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \le x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

Quick Sort — Partitioning (Cont'd)

Pivot: always select last element A[*r*] in A[*p..r*]

array is partitioned into four regions

- region1: A[p..*i*] ≤ pivot
- region2: A[*i+1..j-1*] > pivot
- region3: A[*r*] == pivot (last element)
- region4: A[*j*..r-1] not examined

loop invariant: 1~3

Quick Sort — Partitioning (Cont’d)

idea of partitioning — check the element in region 4, i.e., A[j..r-1], one by one.

- if A[j] < pivot, move the upper boundary of A[p..i] by i=i+1, exchange A[i] vs A[j]
- otherwise, just move second region boundary

Quick Sort — Example

Try to partition the following subarray using PARTITION:

6, 10, 13, 5, 8, 23, 9, 11

What value of q does PARTITION return?

6, 10, 5, 8, 9, **11**, 13, 23 (6 is returned, 11 is the pivot)

Try to partition <6, 10, 5, 8, 9> by yourself

Quick Sort — Partitioning (Cont'd)

Running Time for Partitioning?

- $T(n) = \Theta(n)$ for n elements subarray
  - only have one *for* loop

Quick Sort — Example

What value of *q* does PARTITION return for the following cases?

(1) when all elements in the array A[p..r] have the same value?

(2) when all elements in A[p..r] are distinct and in ascending order?

(3) when all elements in A[p..r] are distinct in descending order?

Quick Sort — Running Time Analysis

```
QUICKSORT($A, p, r$)
1  if $p < r$
2      $q$ = PARTITION($A, p, r$)              $\Theta(n)$
3      QUICKSORT($A, p, q - 1$)               $T(q-1)$
4      QUICKSORT($A, q + 1, r$)               $T(n-q)$
```

Line 3~4: depending on how we partition

$T(n) = T(q-1) + T(n-q) + \Theta(n)$,

where $1 \leq q \leq n$, q is the index of pivot

Quick Sort — Running Time Analysis (Cont'd)

Example 1: Analyze the running time when all elements in the array are identical, such as

5, 5, 5, 5, 5, 5

the last element in the subarray (i.e., A[r]) will be picked as the pivot for every partition (so r is returned as q for every partition)

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Quick Sort Running Time Analysis (Cont'd)

Worst-case partitioning:

occurs when the partitioning routine produces on subproblem with n-1 elements and one with 0 elements. (e.g., identical elements in array)

assume this <u>unbalanced partitioning</u> arises in each recursive call

$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$
$= \Theta(n^2)$

Not better than Insertion Sort! We're unlucky!

Quick Sort — Running Time Analysis (Cont'd)

Example 2: Analyze running time when we split array into half in every partition

q = (p+q)/2 (to be more accurate, the floor)

subproblem sizes: n/2 and n/2-1

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n) \\ &= \Theta(n\lg n) \quad \text{(Master Method case 2 applies)} \end{aligned}$$

We are lucky!

Quick Sort Running Time Analysis (Cont'd)

Best-case partitioning:

occurs when partitioning routine produces 2 subproblems, each of size is n/2 for every split

<u>Balanced partitioning</u> arises in each recursive call

$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

Quick Sort — Running Time Analysis (Cont'd)

Example 3: Analyze running time when PARTITION always produces a 9-to-1 proportional split

- q is located in 9/10 of the length of the array, e.g., if n=100, q is 90

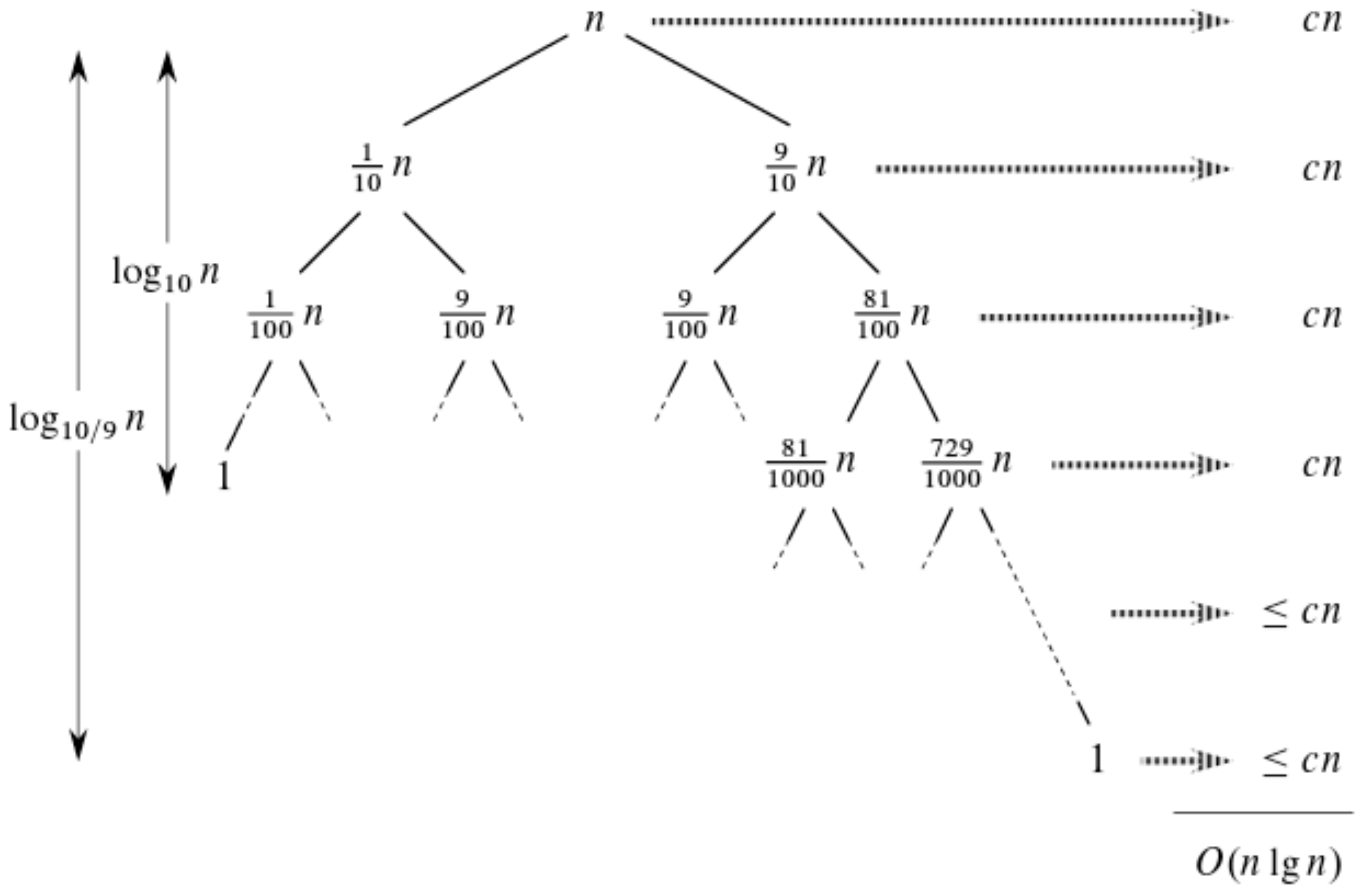
Are we lucky this time? (balanced or unbalanced?)

- subproblem sizes: 9n/10 first half, n/10 second half

$T(n) = T(9n/10) + T(n/10) + \Theta(n)$

we solved a similar recurrence before $T(n) = T(n/3) + T(2n/3) + \Theta(n)$

Quick Sort Running Time Analysis (Cont'd)



Quick Sort Running Time Analysis (Cont'd)

$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$
$$= \Theta(n\lg n)$$

Yes, we are still lucky!

Quick Sort Running Time Analysis (Cont'd)

Balanced partitioning:

any split of <u>constant proportionality</u> yields a recursion tree of depth O(nlgn), where the cost at each level is O(n), e.g., 9-to-1, 99-to-1, 3-to-7

the tree is not balanced, but partitioning is considered as balanced

Quick Sort Running Time Analysis (Cont'd)

Theorem:

$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n)$, where $0<\alpha \leq 1/2$

$T(n) = \Theta(n\lg n)$

e.g.,

1-to-2 split:

$\alpha = 1/3$, $1-\alpha=2/3$, $T(n) = T(n/3)+T(2n/3) + \Theta(n)$

1-to-8 split:

$\alpha = 1/9$, $1-\alpha=8/9$, $T(n) = T(n/9)+T(8n/9) + \Theta(n)$

Quick Sort Running Time Analysis (Cont’d)

Average-case partitioning Intuition:

We may not be lucky or unlucky all the time

quick sort behavior depends on the relative ordering of the values in the array

in the average case, PARTITION produces a mix of “good” and “bad” splits

still O(nlgn), close to best-case running time

HW10

- Due Date: June 24, 11:59pm
- research the applications of an algorithm learned in class (including homework)
    - by reading scholarly academic papers (e.g. ACM, IEEE)
- write a report to summarize findings
    - Introduction, Applications, Conclusions, Reference
    - cite the papers, IEEE format

Probability Basics & Algorithms

Probability Basics

Sample space, events, random variables, expectation (Linearity of Expectation)

Reference:

- Appendix C
- Lehman and Leighton Math pdf (Blackboard), Chapter 21 & 22

Sample space

Sample Space: S {all possible outcomes}

- Algorithm design: S is usually finite (discrete probability)
- each outcome $i \in S$ has a probability, $Pr(i) \geq 0$
- constraint: $\sum_{i \in S} Pr(i) = 1$

Sample space (Cont'd)

Ex1: rolling 2 ordinary, 6-side dice

36 possible outcomes, S = {(1, 1), (2, 1) … (5, 6), (6, 6)}

Pr(i) = 1/36

Ex2: choosing a random pivot in Quicksort

S={1, 2, 3,…, n}

Pr(i) = 1/n for n different choices, equally occur

Events

an **event A** is a subset of S, $A \subseteq S$

the probability of an event A is $\sum Pr(i)$, $i \in A$ (i is an outcome in A)

Events (Cont'd)

Ex1: consider the event "the sum of the two dice is 8", what is the probability of this event?

A={(2, 6), (6, 2), (3, 5), (5, 3), (4, 4)}

Pr{A} = 5/36

Events (Cont'd)

Ex2: consider the event "the chosen pivot gives a split that is better than 25-to-75"

A = {not in the fist quarter, not in the last quarter}, choose (26, 75] elements if we have 100 elements in the array

Pr{A} = n/2 * 1/n = 1/2

Random Variables

a random variable X is a real-valued *function* whose domain is the sample space.

Random Variables (Cont'd)

Ex1: let X be "the sum of the two dice"

- every outcome of two dice uniquely determines the values of X
- X is a function that maps each outcome in S to a number: X(1,1) = 2, …, X(6,6) = 12
- X(x, j) = i + j, can be 2 to 12

Random Variables (Cont'd)

Ex2: let Y be the size of subarray passed to the first recursive call in QUICKSORT

Y is a function that maps each outcome in S to a number: Y(pivot is 1st) = 0, Y(pivot is 2nd) =1, …, Y(pivot is n$^{th)}$ = n-1

Y(pivot index) can be 0 to n-1

Expectation

Let X be a random variable in S, the **expectation of X** is denoted as E[X]: the average value of X, naturally weighted by the probability of the various possible outcomes

$E[X] = \sum_{i \in S} X_i * Pr(i)$, i denotes one possible outcome

mathematically, it is the sum, over everything that could happen, of the value of this random variable (when that outcome occurs) times the probability of that outcome occurring

Expectation — Examples

Ex1: What is the expectation of the value of rolling a single die?

$E[X] = (1+2+3+4+5+6)*1/6 = 3.5$

Expectation — Examples (Cont'd)

Ex2: What is expectation of a random variable X "the sum of two dice"?

E[X] = 2*1/36 + 3*1/36 + 4*3/36 + 5*4/36 + 6*5/36 + 7*6/36 + 8*5/36 + 9* 4/36 + 10*3/36 + 11*2/36 + 12*1/36 = 252/36 = 7

is there an easier way to calculate?

Expectation — Examples (Cont'd)

Ex3: What is the expectation of the size of the subarray passed to the first recursive call in QUICKSORT?

$$E[Y] = (1+2+3+\ldots+n-1) * (1/n) = (n-1)/2$$

Linearity of Expectation

Let $X_1, X_2, \ldots, X_n$ be random variables defined on S (sample space), then

$$E[X_1+X_2+\ldots+X_n] = E[X_1] + E[X_2]+\ldots+E[X_n]$$

$$E[\sum_{j=1}^{n} X_j] = \sum_{j=1}^{n} E[X_j]$$

meaning: It does not matter if we take the sum first and then take the expectation; Or if we take the exception first and then the sum.

Aways work! even when X_js are not independent

Linearity of Expectation — Example

Ex2: What is expectation of a random variable X "the sum of two dice"? (Method 2)

- the expectation of rolling one single die is $E[X_j] = 3.5$
- Let X_1 (X_2) be the number on the first (second) die
- $E[X_1+X_2] = E[X_1] + E[X_2] = 2 * 3.5 = 7$
- much easier than using the definition

Indicator Random Variables

Given a sample space S and an event A, we define the ***indicator random variable***

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases}$$

it is a MAGIC random variable that maps every outcome to either 0 or 1

I{A} partition S into those outcomes in the event, and those outcomes not in the event.

so I{A} is 1 for i∈A; 0 for j∈{S-A}

Indicator Random Variables (Cont’d)

Lemma

For an event A, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Very simple proof

$$E[X_A] = E[I\{A\}] = 1*Pr\{A\} + 0*Pr\{S-A\} = Pr\{A\}$$

Indicator Random Variables (Cont'd)

Example 1: Determine the expected number of heads when flip a fair coin one time

Sample space $S = \{H, T\}$

H: event that head will show up in one flip

$\Pr\{H\} = \Pr\{T\} = 1/2$

indicator random variable $X_H = I\{H\}$: counts the number of heads in one flip

$E[X_H] = \Pr\{H\} = 1/2$

Indicator Random Variables (Cont'd)

Example 2: Determine the expected number of heads in *n* coin flips

X: random variable. number of heads in n flip, not easy to calculate E[X] = $\sum_{k=0}^{n} k \Pr\{X = k\}$

use indicator random variable instead

for i = 1, 2, …, n, define X_i = I {the i^{th} flip results in event H}

so $X = X_1 + X_2 + \ldots + X_n$

$E[X_i] = \Pr\{H\} = 1/2$, for i = 1, 2, …, n

$E[X] = E[X_1+X_2+ \ldots +X_n] = 1/2 * n = n/2$

Probabilistic Analysis and Randomized Algorithms

Probabilistic Analysis

Probabilistic Analysis: the use of probability in the analysis of problems

- analyze the running time of algorithms (most often)
- use the knowledge, or make assumptions, about the distribution of inputs, then compute **average-case running time** (averaging the time over all possible inputs)

Probabilistic Analysis (Cont'd)

When to use it (**not applicable to all problems**)

- applicable: when we have knowledge or can make assumption about distribution of the inputs
  - e.g., nearly sorted input for sorting
- not applicable: when we cannot describe a reasonable input distribution
  - happens quite often
  - either we know little about the input distribution, or we cannot model computationally

Randomized Algorithms

Instead of guessing/assuming input distribution, we enforce a random order (i.e., impose a distribution)

randomized algorithm:

its behavior is determined not only by its input but also by the values produced by a random-number generator

Randomized Algorithms - Ex1

A random-number generator: RANDOM (a, b)

- returns an integer between a and b, inclusive, with each such integer being equally likely
- each integer returned by RANDOM is independent of the integers returned on previous calls
- analogy: (b-a+1)-sided die

Randomized Algorithms - Ex1 (Cont'd)

e.g., What is the expected value for RADOM(3, 7)?

returns one of 3, 4, 5, 6, 7

each with probability 1/5

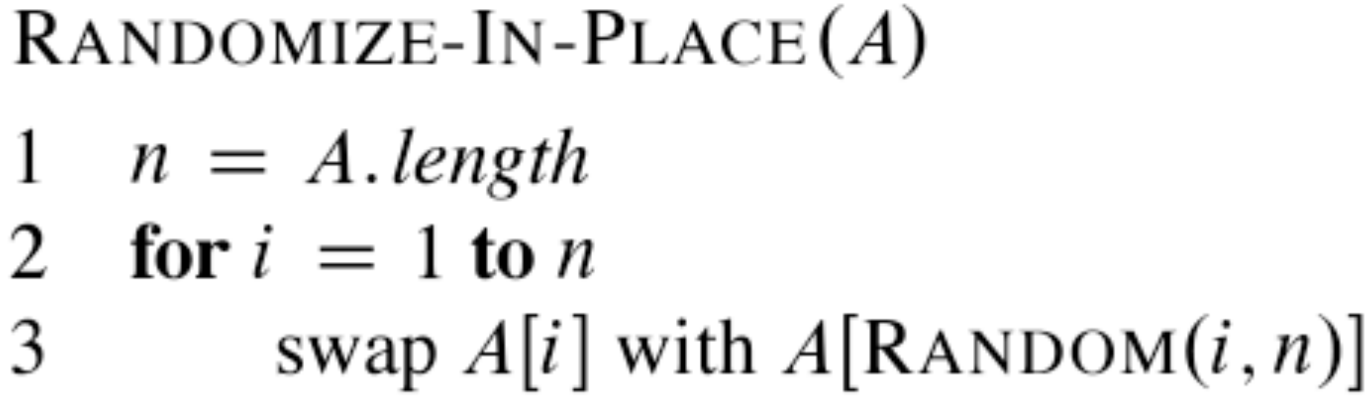
the expected value is 5

Randomized Algorithms - Ex2

Randomly permuting array

- Input: an array A contains elements 1 through n
- Output: a random permutation of the array
- **Uniform Random Permutation**: the procedure is equally likely to produce every permutation of the numbers 1 through n.
- n! permutations, probability 1/n!

Randomized Algorithms — Ex2 (Cont’d)



```
RANDOMIZE-IN-PLACE($A$)
$n = A.length$
for $i = 1$ to $n$
    swap $A[i]$ with $A[\text{RANDOM}(i, n)]$
```

It produces a uniform random permutation

Proof in the book

Randomized Algorithms (Cont'd)

Expected running time: Expectation of running time over the distribution of values returned by the random number generator

Homework 5

Due Date: June 13, 2019 (Th)

BEFORE the class starts

Quick Sort Analysis (Continued)

Quick Sort Worst-case

Worst-case partitioning:

<u>unbalanced partitioning</u> arises in each recursive call

e.g., partitioning routine produces on subproblem with n-1 elements and one with 0 elements (or 0 elements & n-1 elements)

How to beat your enemy?

Suppose that your worst enemy has given you an array to sort with quicksort, knowing that you always choose the rightmost element in each subarray as the pivot, and has arranged the array so that you always get the worst-case split.

How can you defeat your enemy?

Minimize the worst case in Quick sort

- Overall strategy — avoid unbalanced partition
- the current pivot selection does not work well for some inputs, e.g., all elements have the same values
- Pivot strategies, e.g.,:
  - Use the middle Element of the sub-array as the pivot
  - Use the median element of (first, middle, last) to make sure to avoid any kind of pre-sorting

Avoid the worst case in Quick sort

Randomization: Make the chance of worst-case run time equally small for all inputs

- Choose pivot element randomly from range [low..high]
- Initially permute the array

Randomized Quick Sort

makes no assumption about the input distribution

no specific input elicit the worst-case behavior

pivot on random element

Randomized Quick Sort (Cont'd)

Randomized Partition

```
RANDOMIZED-PARTITION$(A, p, r)$
1  $i$ = RANDOM$(p, r)$
2  exchange $A[r]$ with $A[i]$
3  **return** PARTITION$(A, p, r)$
```

Randomized Quick Sort (Cont'd)

Randomized Quick Sort Algorithm

```
RANDOMIZED-QUICKSORT($A, p, r$)
if $p < r$
    $q$ = RANDOMIZED-PARTITION($A, p, r$)
    RANDOMIZED-QUICKSORT($A, p, q - 1$)
    RANDOMIZED-QUICKSORT($A, q + 1, r$)
```

Randomized Quick Sort — worst-case analysis

Worst-case analysis (more rigorously analysis) (for both QuickSort and Randomized Quick Sort)

Upper Bound:

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) ,$$

q: range from 0 to n-1, the subproblem size for the first subarray in which every element is less than or equal to pivot

n-q-1: the subproblem size for the second subarray in which every element is greater than pivot

Substitution method to get upper bound and lower bound

self-study

Randomized Quick Sort — average-case analysis

assume: the values of elements being sorted are distinct

the dominant cost is partitioning

PARTITION removes the pivot element from future consideration each time

how many times can PARTITION be called at most?

n times

running time in PARTITION: some constants plus the number of comparisons in its *for* loop (line 4~6)

Quick Sort — Partitioning Analysis

```
      PARTITION($A, p, r$)
$c_1$  1  $x = A[r]$
$c_2$  2  $i = p - 1$
       3  for $j = p$ to $r - 1$
       4      $\boxed{\textbf{if } A[j] \le x}$
       5          $i = i + 1$
       6          exchange $A[i]$ with $A[j]$
$c_7$  7  exchange $A[i + 1]$ with $A[r]$
$c_8$  8  return $i + 1$
```

line 4: compare pivot with another element

Randomized Quick Sort — average-case analysis(Cont'd)

Let X be the number of comparisons performed in line 4 in PARTITION over the <u>entire</u> execution of QUICKSORT on an n-element array.

The the running time of QUICKSORT is O(n+X)

Proof for Average-case Analysis (next 5 slides)

Quick Sort Performance in Summary

In practice, 3 times faster than merge sort.

most good sorting are based on quick-sort

works well with cache and virtual memory (in space sorting)

Avoid the worst case in Quick sort

- Randomization: Make chance of worst-case run time equally small for all inputs
- Methods
  - Choose pivot element randomly from range [low..high]
  - Initially permute the array

Problem of Hiring (Self Study)