Design and Analysis of Algorithms, MTech-I (1^{st} semester)

Chapter 2: Divide and Conquer: Quicksort

September 21, 2022

Devesh C Jinwala,

Professor in CSE, SVNIT, Surat and Adjunct Professor, IITJammu & Dean (R&C), SVNIT

Department of Computer Science and Engineering, SVNIT, Surat

1. The Quicksort Background
2. The Quicksort Operation and Algorithm
3. The Quicksort Performance Analysis
4. The Randomized Quicksort

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm
- Studied most

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm
- Studied most
- Not difficult to implement - works well with a variety of inputs

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm
- Studied most
- Not difficult to implement - works well with a variety of inputs
- Consumes fewer resources than any other algorithm

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm
- Studied most
- Not difficult to implement - works well with a variety of inputs
- Consumes fewer resources than any other algorithm
- However, is not a stable sorting algorithm.

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm
- Studied most
- Not difficult to implement - works well with a variety of inputs
- Consumes fewer resources than any other algorithm
- However, is not a stable sorting algorithm.
- Almost inplace, uses only a small stack. What is inplace sorting algorithm?

About Quicksort

- An efficient sorting algorithm discovered by C.A.R. Hoare in 1962
- Used most widely than any other sorting algorithm
- Studied most
- Not difficult to implement - works well with a variety of inputs
- Consumes fewer resources than any other algorithm
- However, is not a stable sorting algorithm.
- Almost inplace, uses only a small stack. What is inplace sorting algorithm?
- complexity $O(n\ lg\ n)$ on an average - expected complexity.

About Quicksort....

Name	Average	Worst	Memory	Stable	Method
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$	No	Partitioning
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection

Figure: Sorting algorithms: Complexities

About Quicksort ...

- carefully tuned version is likely to run significantly faster...
- e.g. qsort routine in C++'s STL.
- its average case complexity is considered to be better than that of the Mergesort........ why ?

Quicksort Operation

Operates in two phases

- Partition phase
  - Divides the work into half
- Sort phase
  - Conquers the halves

Partition

- It has hard division and easy combination. . . .
- how does that relate to merge-sort

Quicksort Partitioning

Partitioning

- works on the partitioning of the input array around a pivot

Quicksort Partitioning

Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that

Quicksort Partitioning

Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less

Quicksort Partitioning

Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater

Quicksort Partitioning

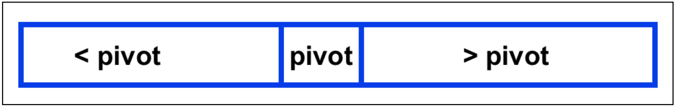
Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater

Quicksort Partitioning

Partitioning

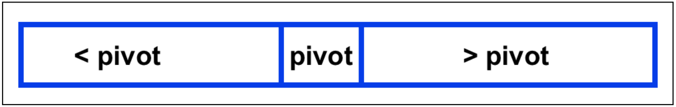
- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater



Quicksort Partitioning

Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater

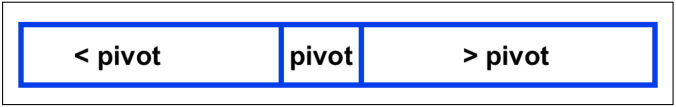


- So, where does the pivot fit in in the ultimate output?

Quicksort Partitioning

Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater

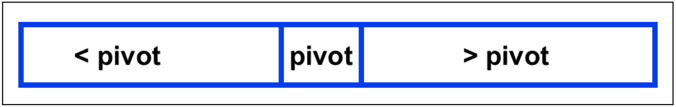


- So, where does the pivot fit in in the ultimate output?
- while partitioning, where do we place/position the pivot element ?

Quicksort Partitioning

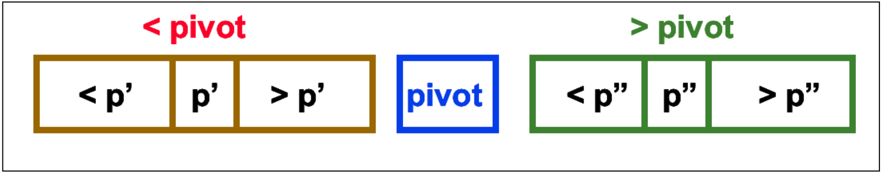
Partitioning

- works on the partitioning of the input array around a pivot
- the aim is to find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater



- So, where does the pivot fit in in the ultimate output?
- while partitioning, where do we place/position the pivot element ?
- most research done on partitioning. . . ..

Conquering



- Apply the same algorithm to each half

The Partition Routine...

Basic Recursive Quicksort

If the size of the list n is 0 or 1, return the list. Otherwise:

1. Choose one of the items in the list as a **pivot**
2. Next, **partition** the remaining items into two disjoint sublists, such that alll the items greater than the pivot are greater than the pivot and all elements less than the pivot are less than the pivot.
3. Finally, return the result of quicksort as the *head* sublist, followed by the pivot, followed by the result of the quicksort of the *tail* sublist.

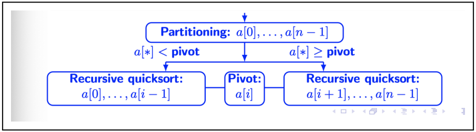


Figure: Quicksort partitioning

The Quicksort Pseudocode

```
Algorithm QUICKSORT(A, low, high)
/* Termination condition! */
1. If (low < high)
2.              pivot = PARTITION(A, low, high);
3.              QUICKSORT(A, low, pivot -1);
4.              QUICKSORT(A, pivot+1, high);
```

- Divide routine ?
- Conquer routine ?

The Partition pseudocode

```
Algorithm PARTITION(A, low, high)
1.      pivot = A[low]
2.      left  = low
3.      right = high+1
4.      while ( TRUE )
5.          do repeat right=right − 1;
6.          until A[right] ≤ pivot
7.          do repeat left=left +1;
6.          until A[left] ≥ pivot
/* right is final position for the pivot */
8.          if left < right
9.                  SWAP(A[left], A[right]);
10.         else
11.                 SWAP(pivot, A[right])
12.                 return right
```

Illustrating Quicksort: Tutorial Problem 1

Work up the algorithm on the following input...

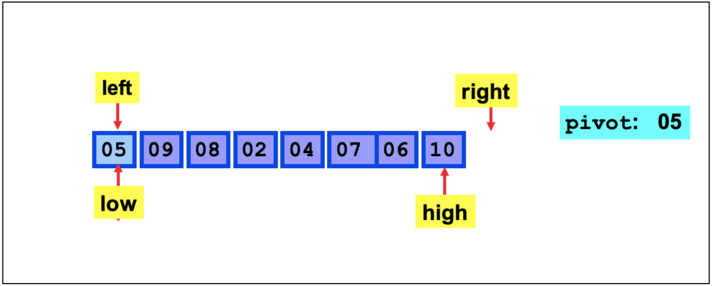


Figure: Run Quicksort - number of swaps & comparisons ?

Illustrating Quicksort:Tutorial Problem 2

Work up the algorithm on the following input.......

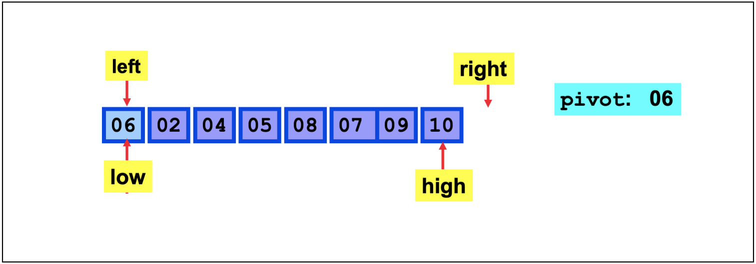


Figure: Run Quicksort - number of comparisons=8 & number of swaps=0??!!!

Illustrating Quicksort: Tutorial Problem 3

Work up the algorithm on the following input.......

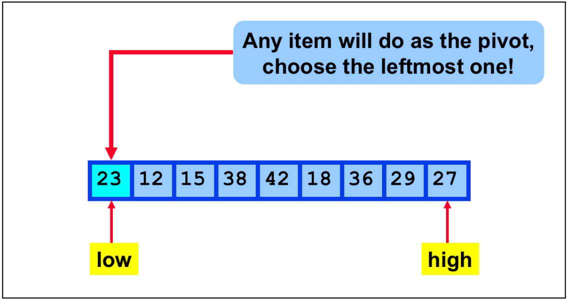


Figure: Run Quicksort - number of swaps=1 & comparisons=10 ?

Illustrating Quicksort: Tutorial Problem 4

Work up the algorithm on the following input

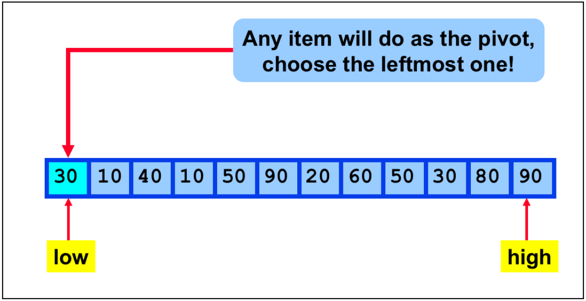


Figure: Run Quicksort - number of swaps & comparisons in each case ?

Illustrating Quicksort: Tutorial Problem 4

Work up the algorithm on the following input

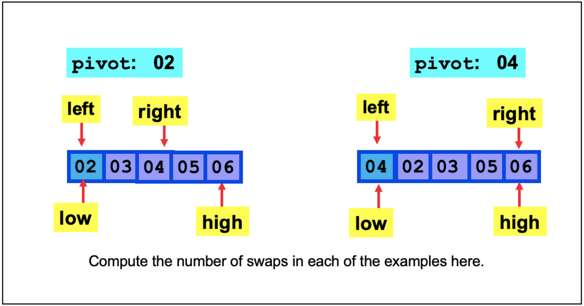


Figure: Quicksort on LHS - number of swaps = 0 (plus a futile last swap) & comparisons=6 ???. Whereas in RHS - number of swaps = 0 & comparisons=6 ????

The Partition pseudocode...:Another approach

```
Algorithm PARTITION(A, low, high)
1.      pivot = A[low]
2.      left  = low+1
3.      right=high
4.      while ( left < right )
5.              while( A[left] < pivot ) left++;
6.              while( A[right]  > pivot ) right--;
7.      if ( left < right ) SWAP(A[left], A[right]);
/* right is final position for the pivot */
8.   SWAP(A[right], pivot);
9.              return right;
```

Dry run of the new PARTITION routine: Homework Assignment

Run the PARTITION routine on the following input instance:
- A(5,9,8,2,4,7,6,10)
- A(23, 12, 15)
- A(10,20,30)

Compute the number of swaps & comparisons ? in each case.

The Partition pseudocode as in the text

```
Algorithm CLRS–PARTITION(A, low, high)
1. pivot = A[high]
2. left = low − 1
3. for right=low to (high −1)
5.              if A[right] <= pivot
6.                      left = left +1
7.                      swap(A[left], A[right])
8. swap(A[left+1], pivot)
9. return left+1
```

Dry run of the PARTITION routine

Run the PARTITION routine on the following input instance:
- A(2,8,7,1,3,5,6,4)
- A(5,9,8,2,4,7,6,10)

The Quicksort Performance

- On one side,

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.
  - The division depends on the position of the pivot element.

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.
  - The division depends on the position of the pivot element.
  - Hence cannot have performance that is provably $O(n \; lg \; n)$ on average case

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.
  - The division depends on the position of the pivot element.
  - Hence cannot have performance that is provably $O(n\ lg\ n)$ on average case
- On the other side

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.
  - The division depends on the position of the pivot element.
  - Hence cannot have performance that is provably $O(n\ lg\ n)$ on average case
- On the other side
  - a D&C algorithm is most efficient when the division is as even as possible.

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.
  - The division depends on the position of the pivot element.
  - Hence cannot have performance that is provably $O(n\ lg\ n)$ on average case
- On the other side
  - a D&C algorithm is most efficient when the division is as even as possible.
  - i.e. quicksort best case occurs when two partitions are of size n/2 each

The Quicksort Performance

- On one side,
  - Unlike mergesort, quicksort may not divide the array into two equal-sized partitions
  - it may produce partitions which range from nearly equal to highly unequal.
  - The division depends on the position of the pivot element.
  - Hence cannot have performance that is provably $O(n\ lg\ n)$ on average case
- On the other side
  - a D&C algorithm is most efficient when the division is as even as possible.
  - i.e. quicksort best case occurs when two partitions are of size n/2 each
  - on an average the performance of quicksort is actually $O(n\ lg\ n)$

The Quicksort Performance Analysis

What is the Recurrence relation ?

- Partition $\Longrightarrow$ Check every item once $\theta(n)$

The above recurrence solves to $\Theta(n\ lg\ n)$. How ?
The best case running time of quick sort is $\Theta(n\ g\ n)$

The Quicksort Performance Analysis

What is the Recurrence relation ?

- Partition $\Longrightarrow$ Check every item once $\theta(n)$
- Conquer $\Longrightarrow$ Conquer data in each division - partition.

The above recurrence solves to $\Theta(n\ lg\ n)$. How ?
The best case running time of quick sort is $\Theta(n\ g\ n)$

The Quicksort Performance Analysis

What is the Recurrence relation ?

- Partition $\Longrightarrow$ Check every item once $\theta(n)$
- Conquer $\Longrightarrow$ Conquer data in each division - partition.
- In general, if the partition is such that it produces two partitions - where one is of size i, then the other would be of size $n - (i - 1)$.

The above recurrence solves to $\Theta(n\ lg\ n)$. How ?
The best case running time of quick sort is $\Theta(n\ g\ n)$

The Quicksort Performance Analysis

What is the Recurrence relation ?

- Partition $\Longrightarrow$ Check every item once $\theta(n)$
- Conquer $\Longrightarrow$ Conquer data in each division - partition.
- In general, if the partition is such that it produces two partitions - where one is of size i, then the other would be of size $n-(i-1)$.
- The generic recurrence of quicksort is as follows:
  $T(n) \;=\; T(i) + T(n-(i-1)) + \theta(n)$

The above recurrence solves to $\Theta(n\ lg\ n)$. How ?
The best case running time of quick sort is $\Theta(n\ g\ n)$

The Quicksort Performance Analysis...

- Partition $\implies$ Check every item once $\theta(n)$
- Conquer $\implies$ Conquer data in each half of roughly size $n/2$
- If we assume partitions of size $\lfloor n/2 \rfloor$ and $\lceil (n-1)/2 \rceil$, then ?
- if we tolerate the sloppiness from ignoring the floor and ceiling and for partitions of from subtracting 1, then ?
- T(n) $= 2T(n/2) + \Theta(n)$
- The above recurrence solves to $\Theta(n\ lg\ n)$. How ?
- The best case running time of quick sort is $\Theta(n\ g\ n)$
- Let us see how.......

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?
- Inferences:

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?
- Inferences:
  1. if the size of the original array is n, then the size of the divided subarrays is definitely going to be less than n.

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?
- Inferences:
  1. if the size of the original array is n, then the size of the divided subarrays is definitely going to be less than n.
  2. As we go down further the subproblems size is getting smaller and so the height of the tree has to be at most n.

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?
- Inferences:
  1. if the size of the original array is n, then the size of the divided subarrays is definitely going to be less than n.
  2. As we go down further the subproblems size is getting smaller and so the height of the tree has to be at most n.
  3. Third, the sum of sizes of children has to be less than or equal to that of the parent.

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?
- Inferences:
  1. if the size of the original array is n, then the size of the divided subarrays is definitely going to be less than n.
  2. As we go down further the subproblems size is getting smaller and so the height of the tree has to be at most n.
  3. Third, the sum of sizes of children has to be less than or equal to that of the parent.
  4. Therefore, the sum of sizes of the children at any level is less than or equal to n - this implies that the work done at each level is O(n).

The Quicksort Performance Analysis: Using Recursion Tree

- Drawing the recursion tree for the invocation of the quicksort
- We start with a problem of size n
- Then, divide it into two parts - one smaller and another larger. See Figure.
- What do we do next ?
- Inferences:
  1. if the size of the original array is n, then the size of the divided subarrays is definitely going to be less than n.
  2. As we go down further the subproblems size is getting smaller and so the height of the tree has to be at most n.
  3. Third, the sum of sizes of children has to be less than or equal to that of the parent.
  4. Therefore, the sum of sizes of the children at any level is less than or equal to n - this implies that the work done at each level is O(n).
  5. Now, the key question is as we saw in Sr no 4 above, what is going to be the height of the tree for a specific input?

The Quicksort Performance Analysis: Recursion Tree

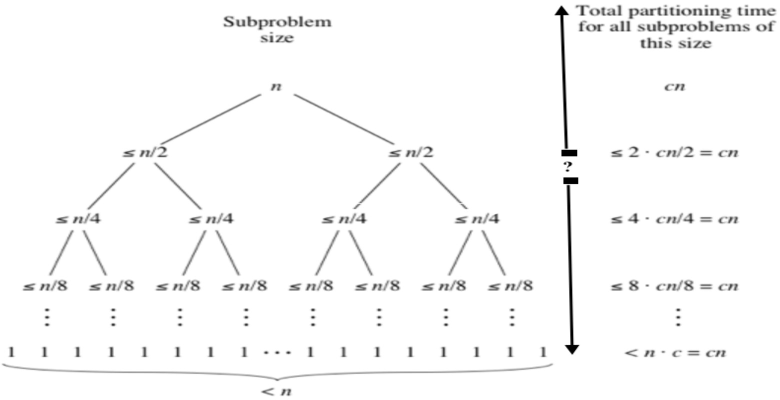


Figure: Recursion Tree: Quicksort Recursion Tree

The Quicksort Performance Analysis: Best case: Recursion Tree

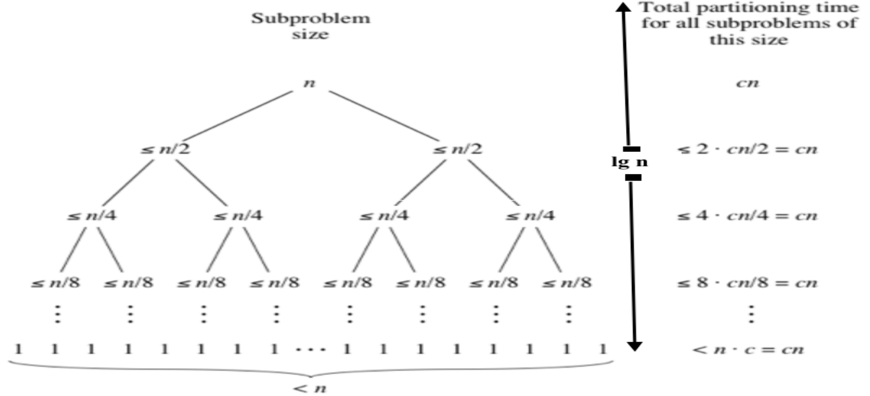


Figure: Recursion Tree: Quicksort Best case

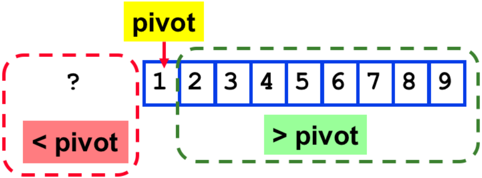
- How can we argue that the height of the tree is lg n?
- But now the question is how to prove this result ?

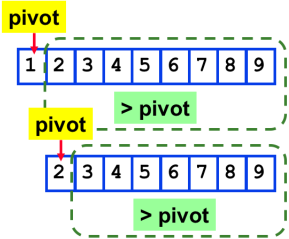
The Quicksort Performance Analysis: Best case

- Prove that the recurrence $T(n) = 2T(n/2) + \Theta(n)$ has the solution $T(n) = \Theta(n \; lg \; n)$
  - Can be done using the Master's theorem - case II applies. Left as an exercise.
  - Can be proven using the Principle of Mathematical Induction - left as an exercise.
- Proving using telescoping- substitution.

The Quicksort Performance Analysis...

- But there is a catch
- Each partition produces a problem of size 0 and one of size (n-1)
- Number of partitions?





The Quicksort Worstcase Performance Analysis

- How is the partitioning appearing now, one that appears to be the worst case?

The Quicksort Worstcase Performance Analysis

- How is the partitioning appearing now, one that appears to be the worst case?
- Well, we have the partitions be such that one is with (n-1) elements and the other is with 0 element

The Quicksort Worstcase Performance Analysis

- How is the partitioning appearing now, one that appears to be the worst case?
- Well, we have the partitions be such that one is with (n-1) elements and the other is with 0 element
- Designing the recurrence relation in this case.....

The Quicksort Worstcase Performance Analysis

- How is the partitioning appearing now, one that appears to be the worst case?
- Well, we have the partitions be such that one is with (n-1) elements and the other is with 0 element
- Designing the recurrence relation in this case.....

The Quicksort Worstcase Performance Analysis

- How is the partitioning appearing now, one that appears to be the worst case?
- Well, we have the partitions be such that one is with (n-1) elements and the other is with 0 element
- Designing the recurrence relation in this case.....

Recurrence Relation on uneven partitioning

$$T(n) = T(n-1) + T(0) + \theta(n)$$

The Quicksort Worstcase Performance Analysis

- How is the partitioning appearing now, one that appears to be the worst case?
- Well, we have the partitions be such that one is with (n-1) elements and the other is with 0 element
- Designing the recurrence relation in this case.....

Recurrence Relation on uneven partitioning

$T(n) = T(n-1) + T(0) + \theta(n)$

- We have to ponder as to what could be T(0)?
- Also, if we assume that the original call takes cn time for some constant c, then $\theta(n) = cn$
- Then the recursion tree takes the form as shown on the next slide... ... ...

The Quicksort Performance: Uneven Partitioning...

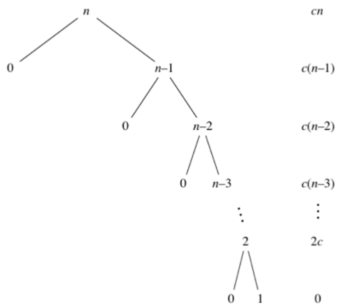


Figure: Quicksort Unbalanced partitioning at each level

Solving the recurrence for the worst case

- Given the recurrence relation as $T(n) = T(n-1) + T(0) + \theta(n)$
- Solving the recurrence....
  - What did we estimate, would be the value of T(0) ?
  - What would be the cost of PARTITION(A, low, high) routine with (n-1) elements in the first recursive call?
  - What would be the cost of PARTITION(A, low, high) routine with (n-2) elements in the second recursive call?
  - Solving the recurrence by iterative method.....

The Quicksort Worst Case

- Prove that the recurrence $T(n) = T(n-1) + \Theta(n) + T(0)$ has the solution $T(n) = \Theta(n^2)$
- Conclusion: Quicksort is as bad as bubble or insertion sort...

The Quicksort Worstcase Performance...

- Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\theta(n^2)$
- Therefore the worst-case running time of quicksort is no better than that of insertion sort.
- Moreover, the $\theta(n^2)$running time occurs when the input array is already completely sorted.
    - a common situation in which insertion sort runs in O(n) time.

The Quicksort Performance: Balanced Partitioning

- Assume that the partitioning is such that the pivot is neither the median of the input array, nor it is skewed i.e. the least or the highest element.

The Quicksort Performance: Balanced Partitioning

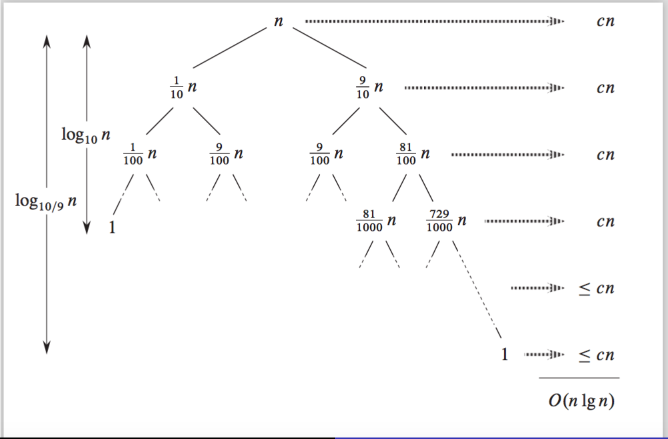
- Assume that the partitioning is such that the pivot is neither the median of the input array, nor it is skewed i.e. the least or the highest element.
- But it is larger than $n/10$ elements and smaller than $n/10$ elements. That is,

The Quicksort Performance: Balanced Partitioning

- Assume that the partitioning is such that the pivot is neither the median of the input array, nor it is skewed i.e. the least or the highest element.
- But it is larger than $n/10$ elements and smaller than $n/10$ elements. That is,
- Assume that the partitioning produces 9-to-1 proportional split on every recursive PARTITION call.

The Quicksort Performance: Balanced Partitioning

- That is, assume that the partitioning produces 9-to-1 proportional split on every recursive PARTITION call.



The Quicksort Performance: Balanced Partitioning...

- Does the partition seen here appear to be balanced ?

The Quicksort Performance: Balanced Partitioning...

- Does the partition seen here appear to be balanced ?
- What is the time complexity ?

The Quicksort Performance: Balanced Partitioning...

- Does the partition seen here appear to be balanced ?
- What is the time complexity ?
- Analysis using the recursion tree

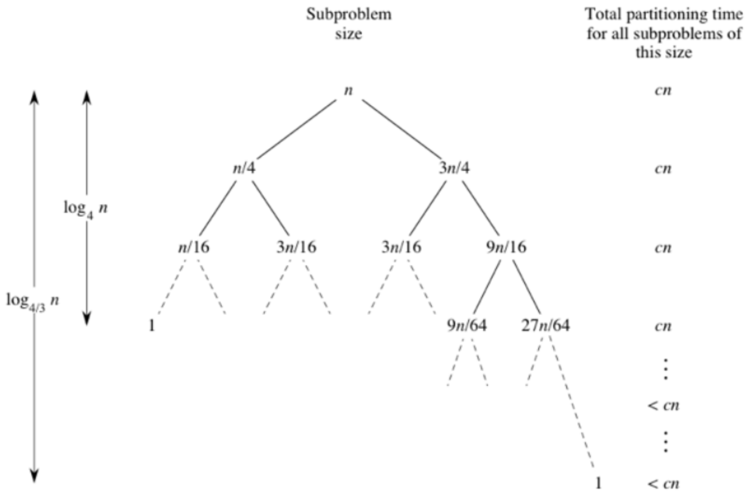
The Quicksort Performance: Balanced Partitioning...

- Does the partition seen here appear to be balanced ?
- What is the time complexity ?
- Analysis using the recursion tree
- What is the recurrence relation ?

The Quicksort Performance: Balanced Partitioning...

- Does the partition seen here appear to be balanced ?
- What is the time complexity ?
- Analysis using the recursion tree
- What is the recurrence relation ?
- What does this recurrence relation solve to ?

The Quicksort Performance: Balanced Partitioning...



The Quicksort Performance: Balanced Partitioning...

The Balanced PArtitioning case

- In fact, any split of constant proportionality yields a recursion tree of depth $\theta(n \; lg \; n)$ where the cost at each level is $O(n)$.
- The running time is therefore $O(n \; lg \; n)$ whenever the split has constant proportionality.

The Quicksort Performance Analysis...

Thus, the Quicksort Performance

- appears to be the same as Heapsort/Mergesort
- quicksort is generally faster
- fewer comparisons or shorter inner loop
- But there is a catch... ... ...

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
- we assume for now that all permutations of the input numbers are equally likely to occur.

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
- we assume for now that all permutations of the input numbers are equally likely to occur.

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
- we assume for now that all permutations of the input numbers are equally likely to occur.
- when we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
- we assume for now that all permutations of the input numbers are equally likely to occur.
- when we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level
  - it is expected that some of the splits will be reasonably well balanced and that some will be fairly unbalanced.

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
- we assume for now that all permutations of the input numbers are equally likely to occur.
- when we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level
    - it is expected that some of the splits will be reasonably well balanced and that some will be fairly unbalanced.

The Quicksort Average Case: Points to be noted

- the behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
- we assume for now that all permutations of the input numbers are equally likely to occur.
- when we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level
  - it is expected that some of the splits will be reasonably well balanced and that some will be fairly unbalanced.
- Exercise 7.2-6 CLRS : Show that about 80 percent of the time PARTITION produces a split that is more balanced than 9 to 1, and about 20 percent of the time it produces a split that is less balanced than 9 to 1.

The Quicksort Average Case: Good and Bad splits distribution...

- for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.

The Quicksort Average Case: Good and Bad splits distribution...

- for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.
- the scenario in the figure shows good and bad splits alternate levels in the tree.

The Quicksort Average Case: Good and Bad splits distribution...

- for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.
- the scenario in the figure shows good and bad splits alternate levels in the tree.
- What is the cost of PARTITION at the root node and at the next node in the hierarchy ?

The Quicksort Average Case: Good and Bad splits distribution...

- for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.
- the scenario in the figure shows good and bad splits alternate levels in the tree.
- What is the cost of PARTITION at the root node and at the next node in the hierarchy ?
- What is the combined cost of PARTITION ?

The Quicksort Average Case: Good and Bad splits distribution...

- for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.
- the scenario in the figure shows good and bad splits alternate levels in the tree.
- What is the cost of PARTITION at the root node and at the next node in the hierarchy ?
- What is the combined cost of PARTITION ?

The Quicksort Average Case: Good and Bad splits distribution...

- for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.
- the scenario in the figure shows good and bad splits alternate levels in the tree.
- What is the cost of PARTITION at the root node and at the next node in the hierarchy ?
- What is the combined cost of PARTITION ?

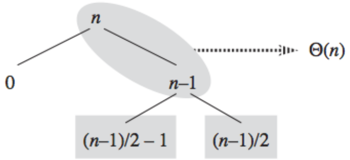


Figure: (a) Bad Split followed by a Good split

The Quicksort Average Case: Good and Bad splits distribution...

- Now consider another average-case execution of PARTITION, in which a single level of partitioning that produces two subarrays of size $(n-1)/2$ *and* $((n-1)/2)-1$ indeed the best case.

The Quicksort Average Case: Good and Bad splits distribution...

- Now consider another average-case execution of PARTITION, in which a single level of partitioning that produces two subarrays of size $(n-1)/2$ *and* $((n-1)/2)-1$ indeed the best case.
- What is the cost of PARTITION at the root node and that at the next node in the hierarchy ?

The Quicksort Average Case: Good and Bad splits distribution...

- Now consider another average-case execution of PARTITION, in which a single level of partitioning that produces two subarrays of size $(n-1)/2$ *and* $((n-1)/2)-1$ indeed the best case.
- What is the cost of PARTITION at the root node and that at the next node in the hierarchy ?
- What is the combined cost of PARTITION ?

The Quicksort Average Case: Good and Bad splits distribution...

- Now consider another average-case execution of PARTITION, in which a single level of partitioning that produces two subarrays of size $(n-1)/2$ *and* $((n-1)/2)-1$ indeed the best case.
- What is the cost of PARTITION at the root node and that at the next node in the hierarchy ?
- What is the combined cost of PARTITION ?

The Quicksort Average Case: Good and Bad splits distribution...

- Now consider another average-case execution of PARTITION, in which a single level of partitioning that produces two subarrays of size $(n-1)/2$ *and* $((n-1)/2)-1$ indeed the best case.
- What is the cost of PARTITION at the root node and that at the next node in the hierarchy ?
- What is the combined cost of PARTITION ?

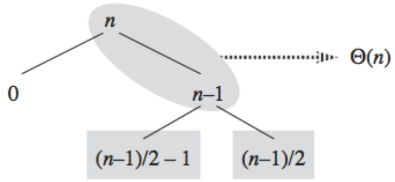


Figure: (b) Bad Split followed by a Good split

The Quicksort Average Case: Good and Bad splits distribution...

- Is the cost of PARTITION in figure (a) worse than that in the figure (b) ?
- Each partition produces a problem of size 0 and one of size (n-1).
- Cost of partitions?

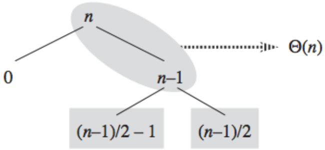


Figure: (a)

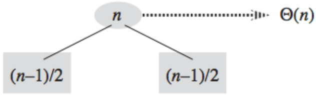


Figure: (b)

The Quicksort Average Case...

Bad split and Good split

- the $\theta(n-1)$ cost of the bad split can be absorbed into the $\theta(n)$ cost of the good split, and the resulting split is good.
- Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone i.e. still $O(n\ lg\ n)$, but with a slightly larger constant hidden by the O-notation.

The Average Case Analysis of Quicksort

- Formal analysis of the average case of the quicksort. To be written on the board.

The Quicksort : Median of three pivot

- Take 3 positions say First, middle, last and choose the median.

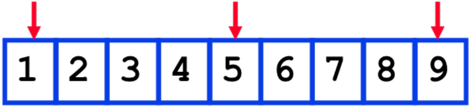


Figure: Pivot is median of three

- What is the median?
- Thus, one gets perfect partitioning yielding $O(n\ lg\ n)$ time complexity
- Since, sorted (or nearly sorted) data is common, median of 3 is a good strategy

MEdian of three does not work always

- However, even any pivot selection strategy could lead to $O(n^2)$ time
- What is the median-of-3 chosen as, in Fig (a) ?
- What is the median-of-3 chosen as, in Fig (b) ?

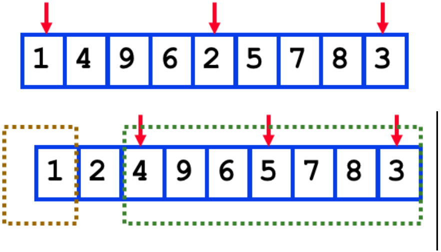


Figure: (a) and (b) Pivot is median of three

The Quicksort: choosing a random pivot

- Choose a pivot randomly

The Quicksort: choosing a random pivot

- Choose a pivot randomly
- Different position for every partition

The Quicksort: choosing a random pivot

- Choose a pivot randomly
- Different position for every partition
- On average, assume the sorted data is divided evenly yielding $O(n\ lg\ n)$ time

The Quicksort: choosing a random pivot

- Choose a pivot randomly
- Different position for every partition
- On average, assume the sorted data is divided evenly yielding $O(n\ lg\ n)$ time
- However, the key requirement is that the pivot choice must take $O(1)$ time

Sorting: Reviewing

Bubble, Insertion

- are $O(n^2)$ sorts
- simple code
- May run faster for small n, n $\tilde{1}0$ (system dependent)

Quick

- is on an average $O(n\ lg\ n)$ sort, but ....
- can deteriorate to $O(n^2)$
- depends on pivot selection
- Better but not guaranteed performance

The Quicksort in comparison

- Quicksort is generally faster - an empirical comparison

n	Quick		Heap		Insert	
	Comp	Exch	Comp	Exch	Comp	Exch
100	712	148	2842	581	2595	899
200	1682	328	9736	9736	10307	3503
500	5102	919	53113	4042	62746	21083

Figure: A typical Empirical Comparison

Quicksort and Heapsort

- Quicksort
  - Generally faster
  - Sometimes $O(n^2)$
  - Better pivot selection reduces probability
  - Used when average good performance is desired
  - Commercial applications, Information systems
- Heap Sort
  - Generally slower
  - Guaranteed $O(n \; log \; n)$
  - Used for real-time systems, where time is a constraint

The Randomized Quicksort

- To eliminate the worst case in quicksort

The Randomized Quicksort

- To eliminate the worst case in quicksort
  - either assume that all the permutations of the input members are equally likely ?!!!!

The Randomized Quicksort

- To eliminate the worst case in quicksort
  - either assume that all the permutations of the input members are equally likely ?!!!!
  - instead of assuming an input distribution, impose a distribution

The Randomized Quicksort

- To eliminate the worst case in quicksort
  - either assume that all the permutations of the input members are equally likely ?!!!!
  - instead of assuming an input distribution, impose a distribution
    - before sorting the input, permute the array elements to ensure every permutation to be likely

The Randomized Quicksort

- To eliminate the worst case in quicksort
  - either assume that all the permutations of the input members are equally likely ?!!!!
  - instead of assuming an input distribution, impose a distribution
    - before sorting the input, permute the array elements to ensure every permutation to be likely
    - does this approach ensure prevention of worst case running time?

The Randomized Quicksort

- To eliminate the worst case in quicksort
  - either assume that all the permutations of the input members are equally likely ?!!!!
  - instead of assuming an input distribution, impose a distribution
    - before sorting the input, permute the array elements to ensure every permutation to be likely
    - does this approach ensure prevention of worst case running time?
    - use the median of the three approach

The Randomized Quicksort

- To eliminate the worst case in quicksort
  - either assume that all the permutations of the input members are equally likely ?!!!!
  - instead of assuming an input distribution, impose a distribution
    - before sorting the input, permute the array elements to ensure every permutation to be likely
    - does this approach ensure prevention of worst case running time?
    - use the median of the three approach
    - use a randomized version of the algorithm

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator
- How is the behaviour (including correctness) is determined ?

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator
- How is the behaviour (including correctness) is determined ?
- output is unpredictable. Why ?

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator
- How is the behaviour (including correctness) is determined ?
- output is unpredictable. Why ?
- output may also vary for the same input, too.

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator
- How is the behaviour (including correctness) is determined ?
- output is unpredictable. Why ?
- output may also vary for the same input, too.
- Two types

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator
- How is the behaviour (including correctness) is determined ?
- output is unpredictable. Why ?
- output may also vary for the same input, too.
- Two types
    - Las Vegas

Randomized Algorithms

- def: is the one which makes use of a randomizer like a random number generator
- How is the behaviour (including correctness) is determined ?
- output is unpredictable. Why ?
- output may also vary for the same input, too.
- Two types
  - Las Vegas
  - Monte Carlo

Why to use Randomized Algorithms?

- useful, when there are many ways in which an algorithm can proceed

Why to use Randomized Algorithms?

- useful, when there are many ways in which an algorithm can proceed
- but it is difficult to determine a unique guaranteed proven way

Why to use Randomized Algorithms?

- useful, when there are many ways in which an algorithm can proceed
- but it is difficult to determine a unique guaranteed proven way
- then, simply choose a random one

Why to use Randomized Algorithms?

- useful, when there are many ways in which an algorithm can proceed
- but it is difficult to determine a unique guaranteed proven way
- then, simply choose a random one
- used when a deterministic algorithm runs much faster on an average than in worst case.

Why to use Randomized Algorithms?

- useful, when there are many ways in which an algorithm can proceed
- but it is difficult to determine a unique guaranteed proven way
- then, simply choose a random one
- used when a deterministic algorithm runs much faster on an average than in worst case.
- we may be able to eliminate the difference between the good and the bad behaviour.

Monte Carlo Randomized Algorithms

- whose output not only differs from run-to-run but may not even be correct for the same input.

Monte Carlo Randomized Algorithms

- whose output not only differs from run-to-run but may not even be correct for the same input.
- employed

Monte Carlo Randomized Algorithms

- whose output not only differs from run-to-run but may not even be correct for the same input.
- employed
  - when no deterministic or probabilistic algorithm is known to solve a problem

Monte Carlo Randomized Algorithms

- whose output not only differs from run-to-run but may not even be correct for the same input.
- employed
  - when no deterministic or probabilistic algorithm is known to solve a problem
  - the probability of the incorrect answer must be low

Monte Carlo Randomized Algorithms

- whose output not only differs from run-to-run but may not even be correct for the same input.
- employed
  - when no deterministic or probabilistic algorithm is known to solve a problem
  - the probability of the incorrect answer must be low
  - guaranteed to run in poly-time, likely to find correct answer.

Monte Carlo Randomized Algorithms

- whose output not only differs from run-to-run but may not even be correct for the same input.
- employed
  - when no deterministic or probabilistic algorithm is known to solve a problem
  - the probability of the incorrect answer must be low
  - guaranteed to run in poly-time, likely to find correct answer.
  - e.g. Contraction algorithm for global min cut.

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer
- guaranteed to find correct answer, likely to run in poly-time.

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer
- guaranteed to find correct answer, likely to run in poly-time.
- may use randomness to guide the search such that correct solution is guaranteed even if unfortunate choices are made.

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer
- guaranteed to find correct answer, likely to run in poly-time.
- may use randomness to guide the search such that correct solution is guaranteed even if unfortunate choices are made.
- alternatively may allow make wrong turns too bringing them to a dead end, in which case no solution given.

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer
- guaranteed to find correct answer, likely to run in poly-time.
- may use randomness to guide the search such that correct solution is guaranteed even if unfortunate choices are made.
- alternatively may allow make wrong turns too bringing them to a dead end, in which case no solution given.
- When employed?

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer
- guaranteed to find correct answer, likely to run in poly-time.
- may use randomness to guide the search such that correct solution is guaranteed even if unfortunate choices are made.
- alternatively may allow make wrong turns too bringing them to a dead end, in which case no solution given.
- When employed?
    - used when a deterministic solution for a problem is already known, but the difference between the worst and the average case is to be bridged. e.g. ??

Las Vegas Randomized Algorithms

- they make probabilistic choices to help guide more quickly to a correct solution
- these algorithms never return a wrong answer
- guaranteed to find correct answer, likely to run in poly-time.
- may use randomness to guide the search such that correct solution is guaranteed even if unfortunate choices are made.
- alternatively may allow make wrong turns too bringing them to a dead end, in which case no solution given.
- When employed?
  - used when a deterministic solution for a problem is already known, but the difference between the worst and the average case is to be bridged. e.g. ??
  - Can always convert a Las Vegas algorithm into Monte Carlo, but no known method to convert the other way

The RANDOMIZED QUICK-SORT(S)

```
Algorithm RANDOMIZED—QUICKSORT(S)
if |S| = 0 return
choose a pivot—element  a_i ϵ S uniformly at random
for each (a ϵ S)
        if (a < a_i)
                put a in S−
        else if (a > a_i)
                put a in S+
RANDOMIZED—QUICKSORTS—)
        output a_i
RANDOMIZED—QUICKSORT(S+)
```

Randomized Algorithms

- Running time.
- Best case. Select the median element as the splitter: quicksort makes θ(n log n) comparisons.
- Worst case. Select the smallest element as the splitter: quicksort makes $\theta(n^2)$ comparisons.
- Randomize. Protect against worst case by choosing splitter at random.
- Intuition. If we always select an element that is bigger than 25% of the elements and smaller than 25% of the elements, then quicksort makes θ(n log n) comparisons.

b l a n k

b l a n k

b l a n k

b l a n k