Sorting Algorithms

Algorithms

Kyuseok Shim
SoEECS, SNU.

Designing Algorithms

- Incremental approaches
- Divide-and-Conquer approaches
- Dynamic programming approaches
- Greedy approaches
- Randomized approaches

You are not going to win at everything in life. Go out there and do your best. When it is over, congratulate the winner-- if it's not you!

By Gail Denvers's father

You are not going to win at everything in life. Go out there and do your best. When it is over, congratulate the winner-- if it's not you!

By Gail Denvers's father

(Gail won a goldmedal at the 1992 Olympic)

Incremental Approach

- *Sorting*: Permuting a sequence of numbers into ascending order
- Insertion Sort Algorithm
  - Works the way many people sort a hand of playing cards
  - Start with an empty left hand and cards face down on the table
  - Remove one card at a time from the table and insert it into the correct position in the left hand
  - To find a correct position for a card, we compare it with each of the cards already in the hand from right to left
  - At all times, the cards held in the left hand are sorted

Insertion Sort Algorithm

- *Sorting*: Permuting a sequence of numbers into ascending order
- Consists of N−1 passes
  - For pass p = 1 through N−1, it ensures that the elements in position 0 through p are in sorted order.
  - Use the fact that the elements 0 through p−1 are already known to be in sorted order.
- It uses an incremental approach!

Insertion Sort Algorithm

```
INSERTION-SORT(A)
for j<-2 to length[A]
    do key <- A[j]
        Insert A[j] into the sorted sequence A[1..j-1]
        i <- j - 1
        while i>0 and A[i]>key
            do A[i+1] <- A[i]
                i <- i-1
            a[i+1] <- key
```

Insertion Sort Algorithm

Original	34	8	64	51	32	21	Position Moved
After j =2	8	34	64	51	32	21	1
After j = 3	8	34	64	51	32	21	0
After j = 4	8	34	51	64	32	21	1
After j = 5	8	32	34	51	64	21	3
After j = 6	8	21	32	34	51	64	4

Order of Growth

- Rate of growth of the running time really interests us
- Thus, we only consider the leading term of a formula since the lower-order terms are relatively insignificant for large n
- We also ignore the leading term's constant coefficient since constant factors are less significant than the arte of growth
- Thus, we write that insertion sort has a worst case running time of $\Theta(n^2)$

Order of Growth

- We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth
  - Due to constant factors and lower-order terms, this evaluation may be in error for small inputs
  - But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run quickly in the worst case than a $\Theta(n^3)$ algorithm

Inversions

- Given 34, 8, 64, 51, 32, 21
- We have 9 inversions:
  - (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), (32, 21)
  - Swapping two adjacent elements (that are out of place) removes one inversion
  - Thus, this is exactly the number of swaps that need too be (implicitly) performed by insertion sort

Insertion Sort Algorithm

- THEOREM 7.1
  - The average number of inversion in an array of N distinct elements is $N(N-1)/4$.
- Proof:

Insertion Sort Algorithm

- THEOREM 7.1
  - The average number of inversion in an array of N distinct elements is $N(N-1)/4$.
- Proof:
  - For any list L, consider L', the list in reverse order.
  - Consider any pair of two elements in the list (x,y), with $y > x$.
  - In exactly one of L and L', this ordered pair represents an inversion
  - The total number of these pairs in a list L and its reverse L' is $N(N-1)/2$.
  - Thus, an average list has half this amount.

Insertion Sort Algorithm

- THEOREM 7.1
  - Any algorithm that sorts by exchanging adjacent elements requires $\Omega(n^2)$
- Proof:
  - Each swap removes only one inversion, so $\Omega(n^2)$ swaps are required.

Divide and Conquer

- This is more than just a military strategy
- It is also a method of algorithm design that has created such efficient algorithms as Merge Sort, Quick Sort
- In terms or algorithms, this method has three distinct steps:
  - Divide: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
  - Recurse: Use divide and conquer to solve the subproblems associated with the data subsets.
  - Conquer: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem.

Merge Sort

- Divide:

  If S has at least two elements, remove all the elements from S and put them into two sequences, $S1$ and $S2$, each containing about half of the elements of S. (i.e. $S1$ contains the first $\lceil n/2 \rceil$ elements and $S2$ contains the remaining $\lfloor n/2 \rfloor$ elements.
- Recurse: Recursive sort sequences $S1$ and $S2$.
- Conquer: Merge the sorted sequences $S1$ and $S2$ into a unique sorted sequence S.

Merge(A,p,q,r)

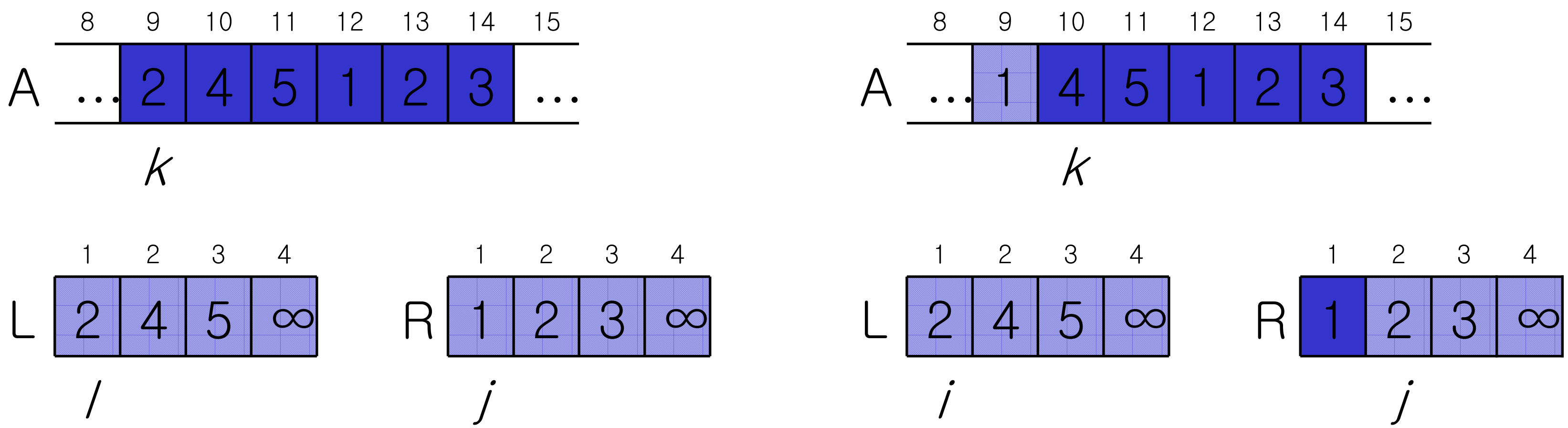
```
n1 <- q – p + 1
n2 <- r – q
create arrays L[1..n1+1] and R[1..n2+1]
for i <- 1 to n1
        do L[i] <- A[p+i-1]
for j <- 1 to n2
        do R[j] <- A[q+j]
L[n1+1] <- ∞
R[n2+1] <- ∞
```

Merge(A,p,q,r)

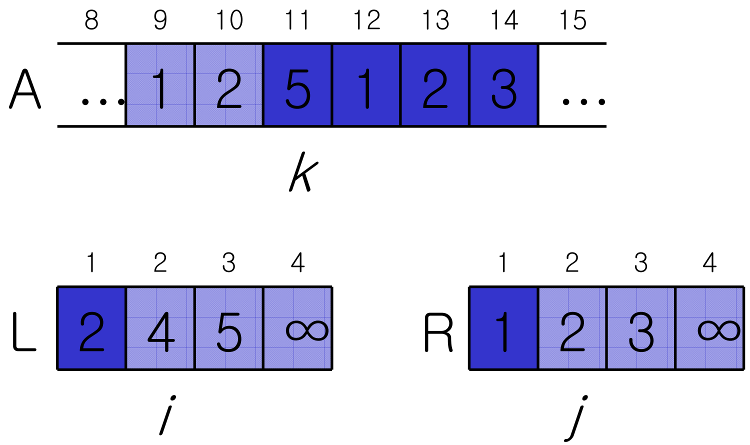
```
i <- 1
j <- 1
for k <- p to r
        do if L[i] <= R[j]
                    then A[k] <- L[I]
                         i <- i + 1
                    else A[k] <- R[j]
                         j <- j + 1
```

Loop Invariant:

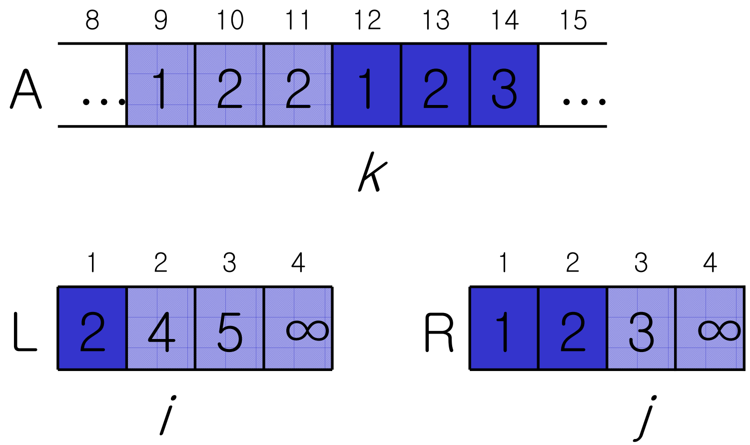
At the start of each iteration for the for-loop above, the subarray A[p..k-1] contains the k-p smallest elements of L[1..n_1+1] and R[1..n_2+1], in sorted order. Moreover, L[I] and R[j] are the smallest elements of their arrays that have not been copied back into A.


8 9 10 11 12 13 14 15
A … 2 4 5 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
R 1 2 3 ∞
j
8 9 10 11 12 13 14 15
A … 1 4 5 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
R 1 2 3 ∞
j

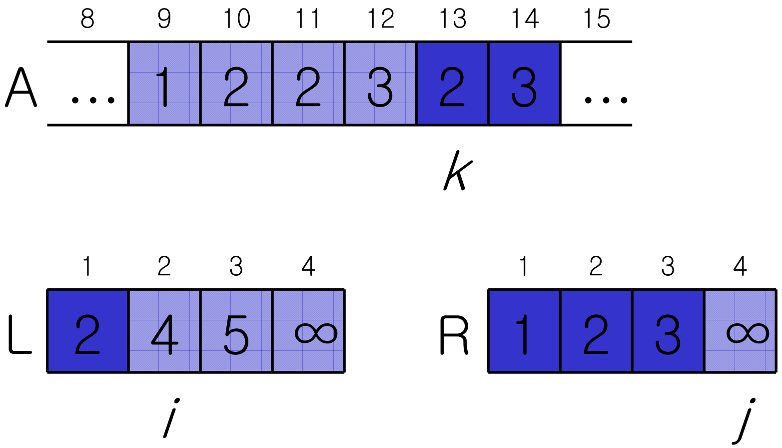
(a) (b)


8 9 10 11 12 13 14 15
A … 1 2 5 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

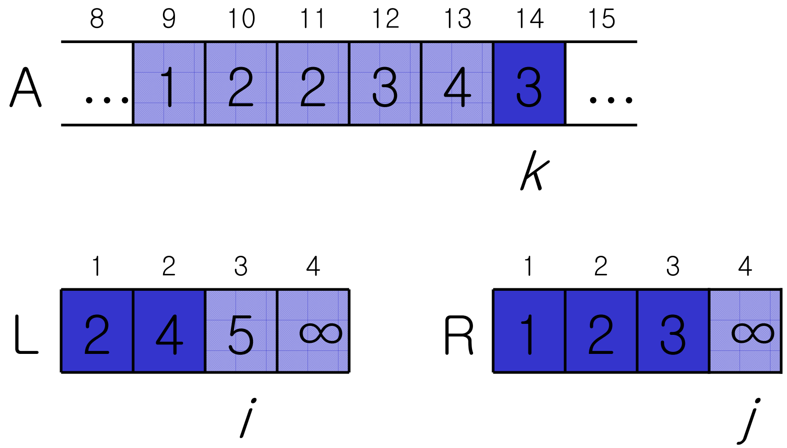
(c)


8 9 10 11 12 13 14 15
A … 1 2 2 1 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

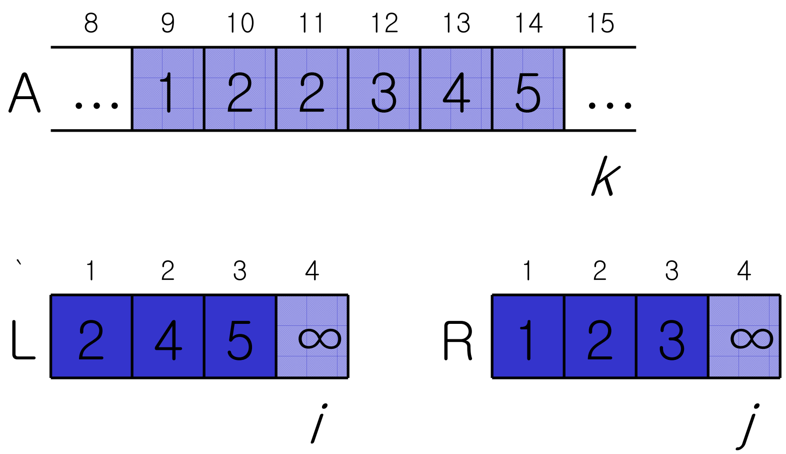
(d)


8 9 10 11 12 13 14 15
A … 1 2 2 3 2 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(e)

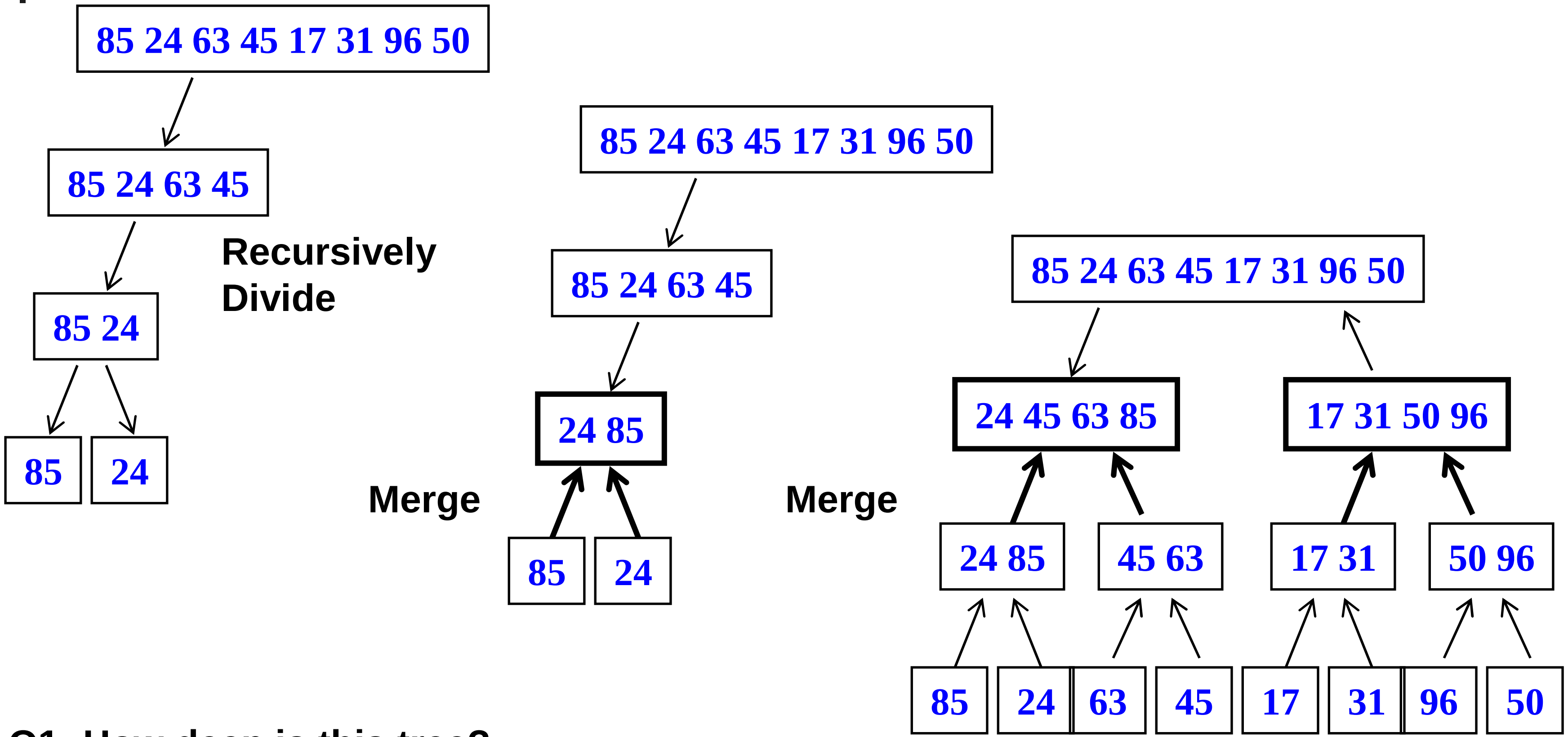

8 9 10 11 12 13 14 15
A … 1 2 2 3 4 3 …
k
1 2 3 4
L 2 4 5 ∞
i
1 2 3 4
R 1 2 3 ∞
j

(f)



(g)

Merge Sort Tree



Q1: How deep is this tree?

Q2: How much memory is needed for merge sort?

Merge Sort

```
MergeSort(A,p,r)
if p < r
then q = floor((p+r)/2)
     MergeSort(A,p,q)
     MergeSort(A,q+1,r)
     Merge(A,p,q,r)
```

- Merge() is the procedure to merge two sorted lists.

Merge Sort Analysis

Recurrence equation :

$$T(1) = 1$$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

$$T(n) = n \log n + n = O(n \log n)$$

Merge Sort

- Merging two half arrays S1, S2 into a full array S requires three pointers, one for S1, another for S2, and the other for S.
- The formal analysis result coincides with the intuitive count of the big Oh, namely, the area taken by the merge sort tree.
- The amount of memory needed for merge sort
  - An extra array

Sorting Algorithms in General

Sorting: Permuting a sequence of numbers into ascending order

O(n^2) Sorting Algorithms:

- Insertion Sort, Bubble Sort

O($n\log n$) Sorting Algorithms

- Heap Sort: Based on Heap data structure
- Quick Sort: Widely regarded as the “fastest” algorithm
- Merge Sort: *Stable* algorithm; if two elements have the same value, then their relative position after sorting is the same

Is it possible to sort faster than O($n\log n$) time?

- Any comparison-based sorting must make at least O($n\log n$) Comparisons in the worst-case
- Linear-Time sorting algorithms for SMALL integers

Quick Sort(cont.)

Given an array A[1...r]

- **Divide:** The array A[1...r] is *partition*ed into two nonempty subarrays A[1...p-1] and A[p+1...r] around the pivot A[p] such that all elements in A[1...p-1] <= A[p] <= all elements in A[p+1...r]

Partition(A,p,r)

```
x ← A[r]
i ← p – 1
for j ← p to r – 1
    do if A[j] ≤ x
            then i ← i + 1
                exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i + 1
```

Quick Sort

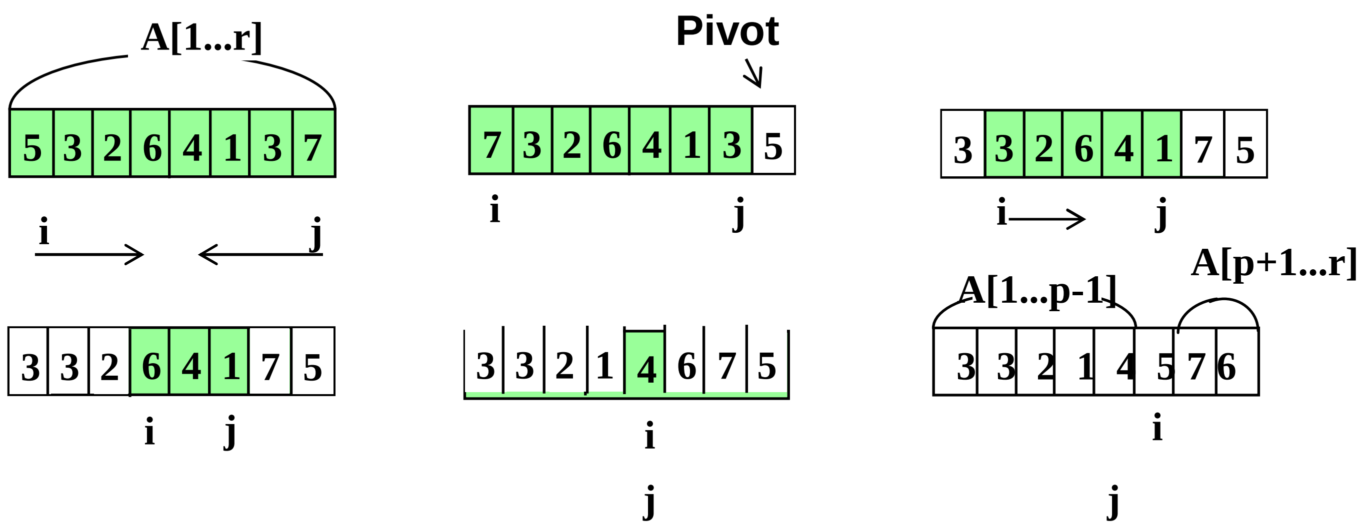
- Conquer: Each of A[1…p] and A[p+1…r] are sorted by recursive calls to Quick sort

```
Quicksort(A,1,r)
  if (1 >= r) return;
  p ← Partition(A,1,r);
  Quicksort(A,1,p-1);
  Quicksort(A,p+1,r);
```

Quick Sort: Partition

Shaded region: not yet partitioned, white region: Partitioned

First, choose the pivot somehow, let's say, it is A[0]=5.
Second, Move the pivot at the end of the array.
Move i to the right until finding the element > the pivot, and
Move j to the left until finding the element < the pivot.



Finally, swap the pivot with the i-th element

Performance of Quick Sort

$T(n) = T(i) + T(n - i - 1) + n(Y(0)) = T(1) = 0)$

Performance depends on the selection of pivot

worst - case partitioning divide $n - 1$ and 1 element

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(1) + \textstyle\sum_{i=2}^{n} i + n \\ &= O(n^2) \end{aligned}$$

best - case partitioning divide $\frac{n}{2}$ and $\frac{n}{2}$ elements

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n \\ &= 2T(\tfrac{n}{4}) + 2n \\ &= O(n \log n) \end{aligned}$$

Performance of Quick Sort-Cont.

Average-case partitioning:

Assume that the size of a partition is equally likely(that is probability is $\frac{1}{n}$)

The average value of $T(i)$ of $T(n-i-1)$ is $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$

$T(n) = \frac{2}{n}[\sum_{j=0}^{n-1} T(j)] + n$

We already know $T(n) = O(n \log n)$ from the average case analysis of unbalanced binary search tree

This average performance requires good selection of pivot!

- Median-of-Partitioning: take the median of the left, right, and center elements in A[l...r]

Selection Problem

- Input: A set of n distinct numbers and a number i with 1 <= i <= n
- Output: The element $x \in A$ that is larger than exactly i-1 other elements of A
- Can be solved in $O(n \log n)$ time by sorting

Quick Selection Algorithm

- Find the k-th smallest element
  - Pick a pivot v in S.
  - Partition S – {v} into S1 and S2
  - If k <= |S1|, then k-th smallest element must be in S1
  - If k = 1 + |S1|, we got the answer
  - Otherwise, the k-th smallest element lies in S2 and it is (k-|S1|-1)st smallest element in S2.

RandomizedSelect(A,p,r,i)

```
if p = r then return A[p]
q <- RandomizedPartition(A,p,r)
k <- q –p +1
if i = k then return A[q]
else if i < k
    return RandomizedSelect(A,p,q–1,i)
else
    return RandomizedSelect(A,q+1,r,i–k)
```

RandomizedSelect(A,p,r,i)

- Worst-case running time is $\Theta(n^2)$
- Average case:
  - RandomizedSelect is equally likely to return any element as the pivot
  - For each k s.t. 1 <= k <= n, the subarray A[p..q] has k elements with probability 1/n
  - Xk = I {subarray A[p..q] has exactly k elements
  - E[Xk] = 1/n

$$T(n) \le \sum_{k=1}^{n} X_k (T(\max(k-1, n-k)) + n)$$

$$= \sum_{k=1}^{n} X_k T(\max(k-1, n-k)) + n)$$

RandomizedSelect(A,p,r,i)

$$E[T(n)]$$

$$\leq E[\sum_{k=1}^{n} X_k T(\max(k-1, n-k)) + n] = \sum_{k=1}^{n} E[X_k T(\max(k-1, n-k))] + n$$

$$= \sum_{k=1}^{n} E[X_k]E[T(\max(k-1, n-k))] + n = \sum_{k=1}^{n} (1/n)E[T(\max(k-1, n-k))] + n$$

$$= \sum_{k=n/2}^{n-1} (2/n)E[T(k)] + n \leq 3cn/4 + c/2 + an = cn - (cn/4 - c/2 - an)$$

Quick Selection Algorithm

- One recursive call contrast to the quicksort algorithm
- Worst case: $\Theta(n^2)$
- Average time complexity: O(n)