Lecture 4: Linear Time Selection

So far we have seen Divide-and-Conquer algorithms that correspond to

$$T(1) = 1, \quad \text{and } \forall n > 1,\ T(n) = 2T(n/2) + O(n)$$
$$\Rightarrow T(n) = \Theta(n \log n)$$

$$T(1) = 1, \quad \text{and } \forall n > 1,\ T(n) = 4T(n/2) + O(n)$$
$$\Rightarrow T(n) = \Theta\left(n^2\right)$$

$$T(1) = 1, \quad \text{and } \forall n > 1,\ T(n) = 3T(n/2) + O(n)$$
$$\Rightarrow T(n) = \Theta\left(n^{\log_2 3}\right)$$

We will now see a D-a-C algorithm corresponding to

$$T(1) = 1, \quad \text{and } \forall n > 1,\ T(n) = T(n/5) + T(7n/10) + O(n)$$
$$\Rightarrow T(n) = \Theta(n)$$

Intuition

For $\alpha_1, \alpha_2, \ldots, \alpha_k > 0$,
given a divide and conquer relation

$$T(n) \leq \sum_{i=1}^{k} T(\alpha_i n) + \Theta(n)$$

Then

- If $\sum_{i=1}^{k} \alpha_i = 1$ then $T(n) = \Theta(n \log n)$
  e.g., $T(n) = 2T(n/2) + \Theta(n)$

- If $\sum_{i=1}^{k} \alpha_i > 1$ then $T(n) = \Theta(n^\beta)$ some $\beta > 1$
  e.g., $T(n) = 3T(n/2) + \Theta(n)$

- If $\sum_{i=1}^{k} \alpha_i < 1$ then $T(n) = \Theta(n)$
  e.g., $T(n) = T(n/5) + T(7n/10) + \Theta(n)$

Outline of this Lecture

- Selection problem and a $\Theta(n \log n)$ solution.
- Review of routine partition(A, p, r) used in Quicksort.
- Quick review of Quicksort
- Linear Time selection

Readings

- **Quicksort and Partition:**
  CLRS, pages 144-53.

- **Linear Time Selection:**
  CLRS, pages 185-195.

- **Review of Divide and conquer recurrences (and the master method for solving them):**
  CLRS, Chapter 4.

The Selection Problem

$Sel(A, i)$:

Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, and an integer i, $1 \leq i \leq n$, find the ith smallest element. When $i = \lceil n/2 \rceil$, this is called the median problem.

Example: Given $\langle 1, 8, 23, 10, 19, 33, 100 \rangle$, the 4th smallest element is 19.

Question: How do you solve this problem?

First Solution: Selection by Sorting

Step 1: Sort the elements in ascending order with any algorithm of complexity $O(n \log n)$.

Step 2: Return the ith element of the sorted array.

The complexity of this solution is $\Theta(n \log n)$.

Question: Can we do better?

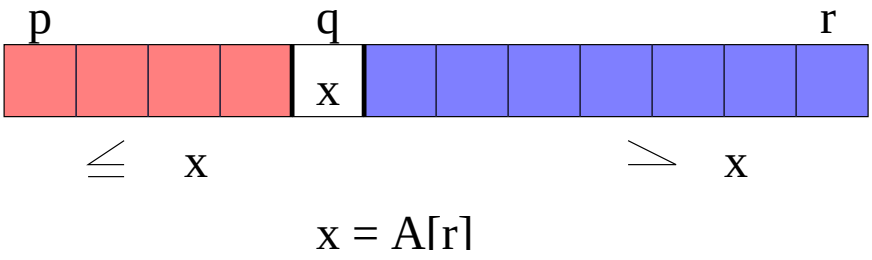
Answer: YES, but we need to recall Partition(A, p, r) used in Quicksort!

Recall of Partition(A, p, r)

Definition: Rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that

$$A[u] \leq A[q] < A[v]$$

for any $p \leq u \leq q-1$ and $q+1 \leq v \leq r$.



(1) x is the original value found in $A[r]$.

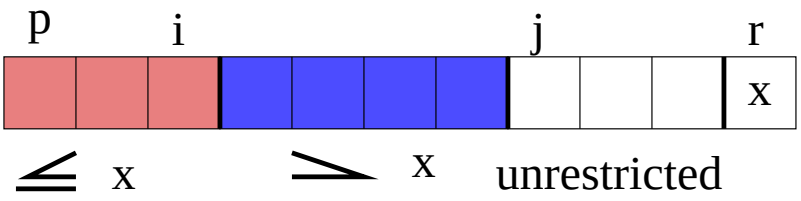
x is the pivot of the algorithm

(2) After Partition(A, p, r) ends we know location of x

This tells us how many items are larger than x and how many are smaller than x.

(3) Running time is $\Theta(p-r)$.

The Idea of Partition(A, p, r)

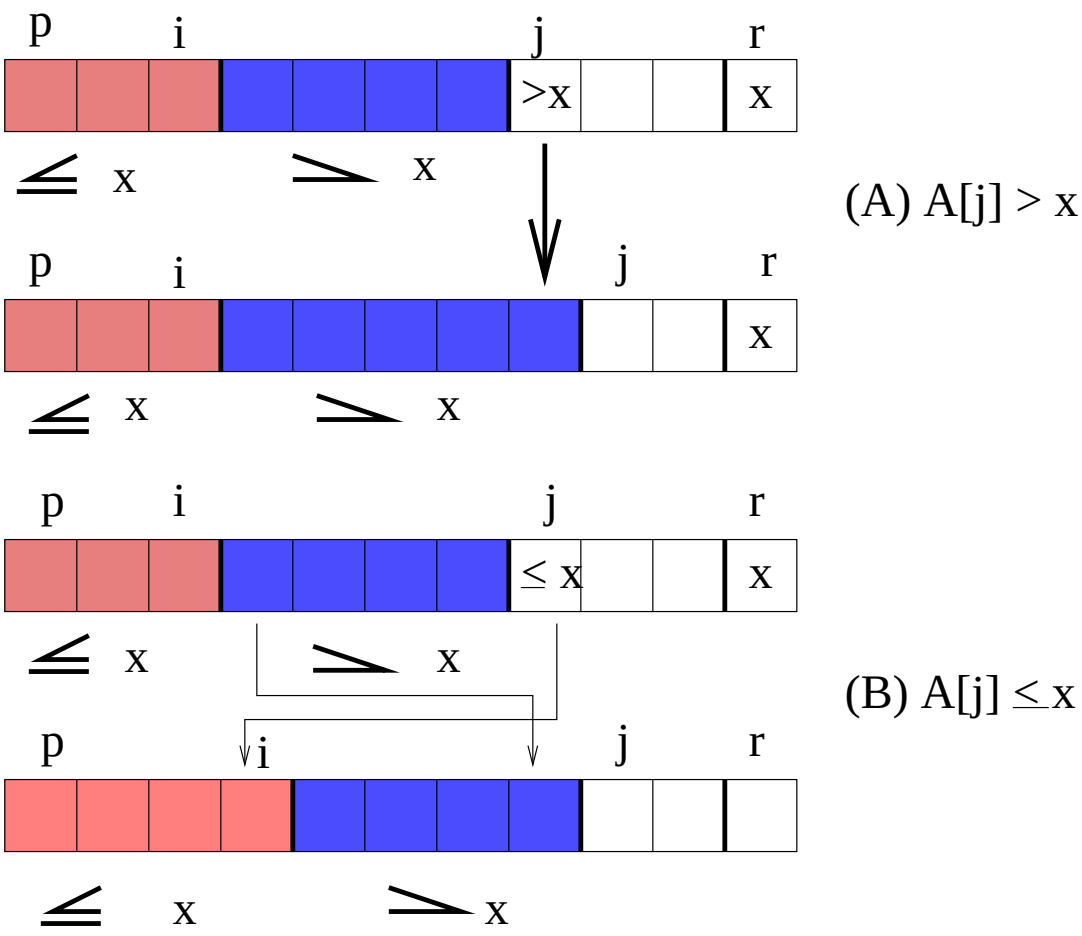


(1) Initially $(i, j) = (p - 1, p)$.

(2) Increase j by 1 each time to find a place for $A[j]$.
At the same time increase i when necessary.

(3) The procedure stops when $j = r$.

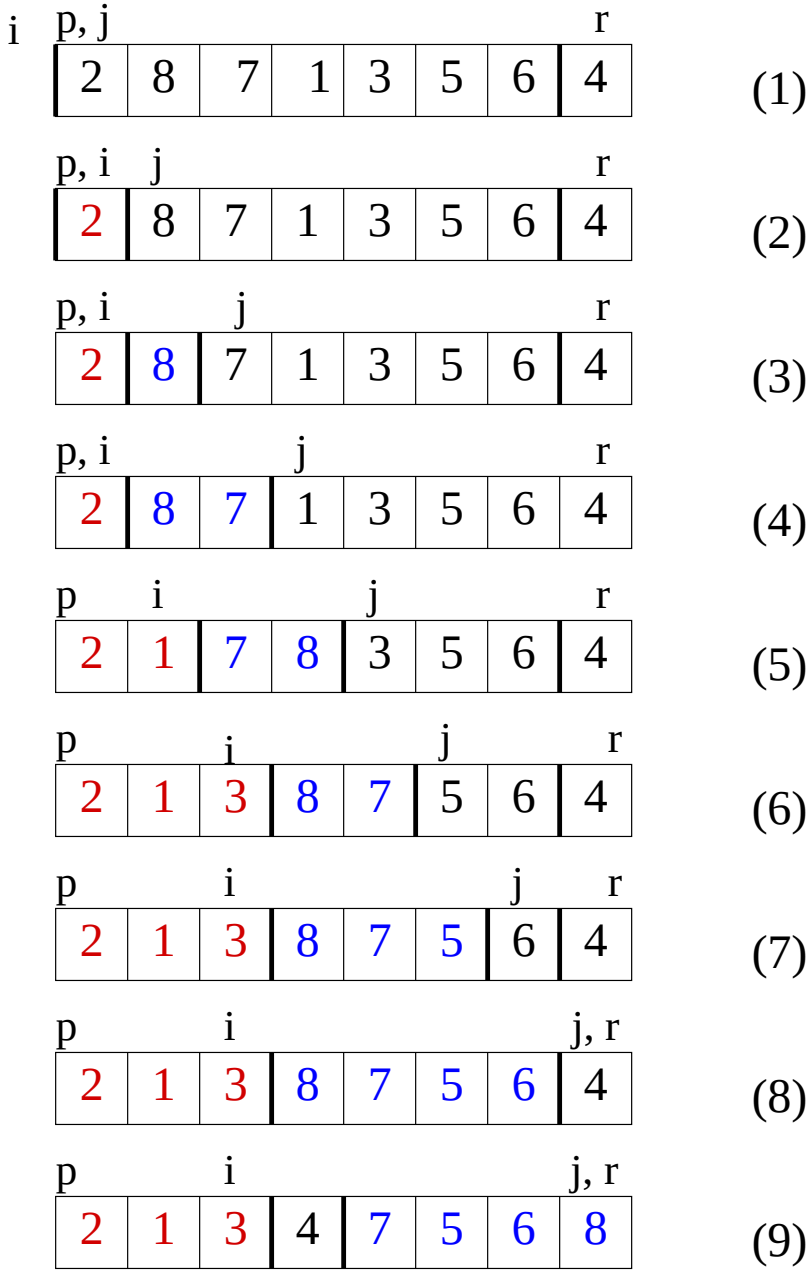
One Iteration of the Procedure Partition



(A) Only increase j by 1.

(B) $i \leftarrow i + 1$. $A[i] \leftrightarrow A[j]$. $j \leftarrow j + 1$.

The Operation of Partition(A, p, r): Example



The Partition(A, p, r) Algorithm

Partition(A, p, r**):**

$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$ (put pivot in position)
return $i + 1$ ($q = i + 1$)

The Running Time of Partition(A, p, r)

comparison of array elements

assignment, addition, comparison of loop variables

Partition(A, p, r**):**

$x = A[r]$	1
$i = p - 1$	1
for $j = p$ to $r - 1$	$2(r - p)$
if $A[j] \leq x$	$(r - p)$
$i = i + 1$	$\leq (r - p)$
exchange $A[i] \leftrightarrow A[j]$	$\leq 3(r - p)$
exchange $A[i + 1] \leftrightarrow A[r]$	3
return $i + 1$	1

Total: $(r - p)$ and $\leq \{6(r - p) + 6\}$

Running time is $\Theta(r - p)$, that is, linear in the length of the array $A[p..r]$.

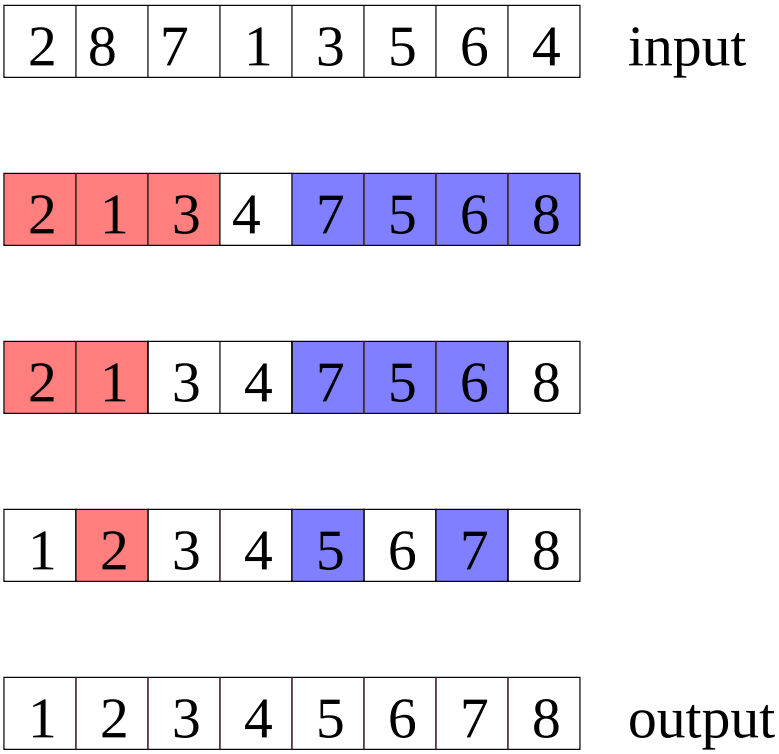
Review of the Quicksort Algorithm

Quicksort(A, p, r)

- **if** $p < r$
- $q =$ Partition(A, p, r)
- Quicksort$(A, p, q - 1)$
- Quicksort$(A, q + 1, r)$

To sort an array $A[1..n]$, call Quicksort$(A, 1, n)$.

Example of Quicksort



Quicksort by recusively
calling Partition(A, p, r)

Running Time of Quicksort

Worst Case: $T(n) = \Theta(n^2)$.

Average Case: $T(n) = O(n \log n)$.

Remark: This is a review only and we do not give the running time analysis.

Exercise: Let $Q(n)$ denote the average number of comparisons of array elements done by the Quicksort algorithm.
Explain why

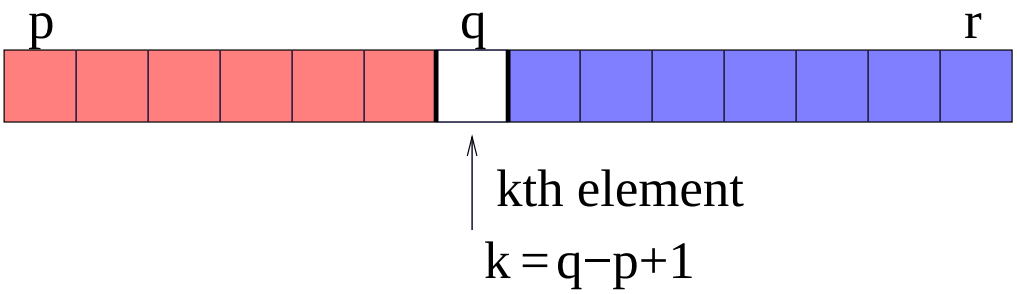
$$Q(n) = n - 1 + \frac{1}{n} \sum_{k=1}^{n} \{Q(n-k) + Q(k-1)\}.$$

Show that

$$Q(n) = 2(n+1)H_n - 4n.$$

A first attempt at a Selection Algorithm

The Idea: To find the ith item in $A[p \ldots r]$ where $1 \leq i \leq r - p + 1$ first find $q = \text{Partition}(A, p, r)$.



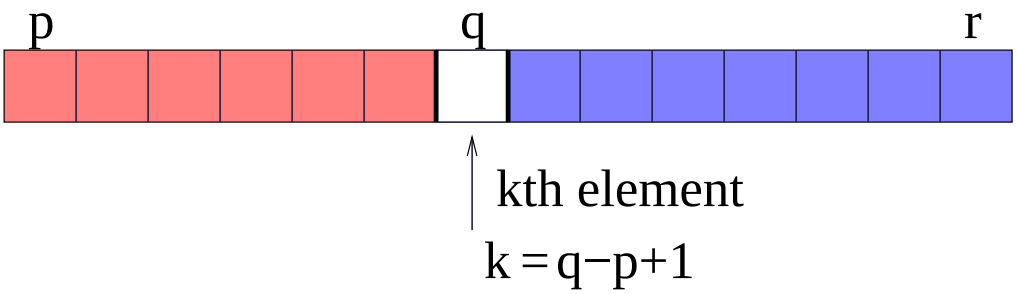
Case 1: $i = k$, pivot is the solution.

Case 2: $i < k$, the ith smallest element in $A[p..r]$ must be the ith smallest element in $A[p..q-1]$.

Case 3: $i > k$, the ith smallest element in $A[p..r]$ must be the $(i-k)$th smallest element in $A[q+1..r]$.

If necessary, recursively call the same procedure on the subarray:

BadSelect(A, p, r, i**),** $1 \leq i \leq r - p + 1$



Find the ith item in $A[p \ldots r]$

i restricted to $1 \leq i \leq r - p + 1$.

if $p == r$

    return $A[p]$

$q =$ Partition(A, p, r) uses $\Theta(p - r + 1)$ time

$k = q - p + 1$

if $i == k$ the pivot is the answer

    return $A[q]$

else if $i < k$

    return BadSelect$(A, p, q - 1, i)$

else

    return BadSelect$(A, q + 1, r, i - k)$

Badselect uses $\Theta(n^2)$ time in worst case!

Why?

Badselect works badly when pivot is based at “wrong end” of the array.

If we could always somehow guarantee that the pivot was the median (middle elements) then at each step the size of the subarray called would be reduced by $1/2$ and the running time of the algorithm would be at most

$$n+\frac{n}{2}+\frac{n}{4}+\cdots+\frac{n}{2^{\lfloor \log_2 n \rfloor -1}}+\frac{n}{2^{\lfloor \log_2 n \rfloor}}+1 = O(n).$$

If we could somehow guarantee that the pivot was near the median then we could guarantee an $O(n)$ running time.

The Linear Time Algorithm Algorithm for $Sel(A, p, r, i)$

1. Divide the $n = p - r + 1$ items into $\lceil n/5 \rceil$ sets in which each, except possibly the last, contains 5 items. $O(n)$

2. Find *median* of each of the $\lceil n/5 \rceil$ sets. $O(n)$

3. Take these $\lceil n/5 \rceil$ medians and put them in another array. Use *Sel()* to *recursively* calculate the *median* of these medians. Call this x. $T(n/5)$

4. Partition the original array using x as the pivot. Let q be index of x, i.e., x is the $k = q - p + 1$'st smallest element in original array. $O(n)$

5. If $i = q$ return x
   If $i < q$ return $Sel(A, p, q - 1, i)$.
   If $i > q$ return $Sel(A, q + 1, r, i - q)$.
   $T(\max(q - p, r - q))$

Termination condition:

If $n \leq 5$ sort the items and return the ith largest.

The algorithm returns the correct answer because lines 4 and 5 will always return correct solution, no matter which x is used as pivot.

The reason for lines 1, 2, and 3 is to guarantee that x is "near" the center of the array.

How many elements in A are greater (less) than x?. Answer (proven next page): At least

$$\frac{3n}{10} - 6.$$

Assuming that $T(n)$ is non-decreasing this implies that time used by step 5 is at most

$$T\left(\frac{7n}{10}\right) + 6.$$

Lemma: At least

$$\frac{3n}{10} - 6$$

elements are greater (less) than x.

Proof: We assume that all elements are distinct (not needed but makes the analysis a bit cleaner).

At least 1/2 of the $\lceil \frac{n}{5} \rceil$ medians in step 2 are *greater* than x.

Ignoring the group to which x belongs and the (possibly small) final group this leaves $\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2$ groups whose medians are greater than x.

Each such group has *at least* 3 items greater than x. Then, number of items greater than x is *at least*

$$3\left(\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Analysis of number less than x is exactly the same!

Running Time of Algorithm

Assume any input with $n \leq 140$ uses $O(1)$ time.

Let a be such that Steps 1,3,4 need at most an time.

Assume that $T(n)$ is non-decreasing. Then

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + an & \text{if } n > 140 \end{cases}$$

We will show, by induction that $T(n) \leq cn$.
Choose c large enough that
$\forall n \leq 140, T(n) \leq cn$.
By induction hypothesis

$$\begin{aligned} T(n) &\leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + an \\ &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

Have already seen that

$$T(n) \leq cn + (-cn/10 + 7c + an).$$

We want to show that $T(n) \leq cn$ so we would be finished if, $\forall n \geq 140$

$$\begin{aligned} 0 &\geq -cn + 70c + 10an \\ &= -c(n - 70) + 10an \end{aligned}$$

or

$$c \geq 10a(n/(n - 70)).$$

Since $n \geq 140$ we have $n/(n - 70) < 2$ so this will be true for any $c \geq 20an$ and we have shown that $T(n) \leq cn$ for all $n \geq 140$ and

$$T(n) = O(n).$$

Review

In this lecture we have

- Reviewed partition and quicksort.
- Reviewed different cases of divide-and-conquer recurrences
- Derived a linear time selection algorithm.

Note: In practice, the constant in the $O(n)$ running time of our selection algorithm is rather high so people use the *randomized* selection algorithm described on pages 185-89 of CLRS.

Review of Divide-and-Conquer

In this section we have learnt how to apply the technique of divide-and-conquer to design of efficient algorithms.

In the simplest version of D-a-C, e.g., maximum contiguous subarray, mergesort, the algorithm partitions the data into (almost) equal sized subinstances, solves the problem on the subinstances, and then combines those solutions together.

In some more advanced cases, e.g., polynomial multiplication, the idea is not to *partition* the data but to create smaller (usually almost equal) sized subproblems that are solved.

In even more advanced cases, e.g., selection, the idea is to observe that the problem can be solved if we are able to solve smaller sized-subproblems. These subproblems might no longer be found by simple data-splitting, though.