

CPSC 311 Lecture Notes

Sorting and Order Statistics (Chapters 6-9)

Acknowledgement: Parts of these course notes are based on notes from courses given by Jennifer Welch at Texas A&M University.

Sorting

The Sorting Problem:

input: a collection of n data items $\langle a_1, a_2, \dots, a_n \rangle$ where each data item has a *key* drawn from a *linearly ordered* set (e.g., ints, chars)

output: a permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input items arranged in increasing order of their keys, i.e., so that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

The Data:

- In practice, one usually sorts **‘records’** according to their **key** (the non-key data is called **satellite data**).
- Sometimes, if the records are large, we sort an array of pointers to the records, rather than moving the records themselves.
- So we can concentrate on the sorting problem itself, we'll usually assume the records have no satellite data (i.e., are keys only).

Sorting Algorithms

Sorting Algorithms

- A sorting algorithm is **in place** if only a constant number of elements of the input array are ever stored outside the array.
- A sorting algorithm is **comparison based** if the only operation we can perform on keys is to **compare two keys**. The comparison sorts (insertion, merge sort, heapsort, quicksort) determine the sorted order of an input array by comparing elements.

Comparison Based Sorts				
Algorithm	Running Time			in place
	worst-case	average-case	best-case	
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$	yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	no
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	yes
Non-Comparison Based Sorts				
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	no
Radix Sort	$O(d(n + k'))$	$O(d(n + k'))$	$O(d(n + k'))$	no
Bucket Sort		$O(n)$		no

The non-comparison based sorts (counting, radix, bucket) sort numbers by means other than comparing two elements.

- Counting Sort: assumes input elements are in a range of $[1, 2, \dots, k]$, and uses array indexing to count the number of elements of each value.
- Radix Sort: an **integer** sorting algorithm, assumes each integer consists of d digits, and each digit is in the range $[1, 2, \dots, k']$.
- Bucket Sort: requires advance knowledge of input **distribution** (sorts n numbers uniformly distributed in range in $O(n)$ time).

Priority Queue Data Structures

Definition: A **priority queue** is a data structure for maintaining a set S of elements, each with an associated **key**. It supports the following operations:

- insert(S, x) inserts element x into S
- max(S) returns element of S with largest key
- extract-max(S) removes and returns element of S with largest key

Sorting with a priority queue

PRIORITYQUEUE-SORT(A)

$S := \emptyset$

for $i := 1$ **to** n /**insert elts into PQ**/

 insert($S, A[i]$)

for $i := n$ **downto** 1 /**remove elts from PQ in sorted order**/

 SortedA[i] := extract-max(S)

Heap Sort is an algorithm like PRIORITYQUEUE-SORT, which uses a **heap data structure** as the priority queue.

Heap Sort

HEAPSORT(A)

```

BuildMaxHeap( $A$ ) /** put all elts in heap **/
for  $i := n$  downto 2
  swap( $A[1]$ ,  $A[i]$ ) /** ‘extract-max’ **/
  heap-size[ $A$ ] := heap-size[ $A$ ] – 1
  MaxHeapify( $A$ , 1) /** fix up the heap **/

```

We'll next see that

- BUILDMAXHEAP(A) takes $O(|A|)$ time
- MAXHEAPIFY(A , 1) takes $O(\log |A|)$ time

Running Time of HeapSort

- 1 call to BUILDMAXHEAP()
 - ⇒ $O(n)$ time
- $n - 1$ calls to MAXHEAPIFY()
 - ⇒ each takes $O(\log n)$ time
 - ⇒ $O(n \log n)$ time for all of them

Grand Total: $O(n \log n)$

(Binary) Heaps

heap: a *conceptual* implementation

- complete binary tree, except may be missing some rightmost leaves on bottom level
- each node contains a key
- a node's key is \geq its children's keys (**heap property**)

binary tree: an array implementation

- root is $A[1]$
- for element $A[i]$
 - left child is in $A[2i]$
 - right child is in $A[2i + 1]$
 - parent is in $A[\lfloor i/2 \rfloor]$

heap: an *array* implementation

- store heap as a binary tree in an array
- **heapsize** is number of elements in heap
- **length** is number of elements in array (maybe more than are currently in heap)

Heapify: Maintaining the Heap Property

MAXHEAPIFY(A, i)

- *assume*: subtrees rooted at left and right children of $A[i]$ are heaps ($A[2i]$ and $A[2i + 1]$)
- but... subtree rooted at $A[i]$ might not be a heap (that is, $A[i]$ may be smaller than its left or right child)
- MAXHEAPIFY(A, i) will cause the value at $A[i]$ to “float down” in the heap so that subtree rooted at $A[i]$ becomes a heap

MAXHEAPIFY(A, i)

```

left :=  $2i$  /**index of leftchild of  $A[i]$ */
right :=  $2i + 1$  /**index of rightchild of  $A[i]$ */
if left ≤ heapsize( $A$ ) and  $A[\textit{left}] > A[i]$ 
    then largest := left
    else largest :=  $i$ 
if right ≤ heapsize( $A$ ) and  $A[\textit{right}] > A[\textit{largest}]$ 
    then largest := right
if largest ≠  $i$ 
    then swap( $A[i], A[\textit{largest}]$ )
        MAXHEAPIFY( $A, \textit{largest}$ )

```

MaxHeapify – Running Time

```

MAXHEAPIFY(A, i)
  left := 2i /**index of leftchild of A[i]**/
  right := 2i + 1 /**index of rightchild of A[i]**/
  if left ≤ heapsize(A) and A[left] > A[i]
    then largest := left
    else largest := i
  if right ≤ heapsize(A) and A[right] > A[largest]
    then largest := right
  if largest ≠ i
    then swap(A[i], A[largest])
      MAXHEAPIFY(A, largest)

```

Running Time of MAXHEAPIFY

- everything is $\Theta(1)$ time except the recursive call
- in worst-case, last row of binary tree is half empty
 - children's subtrees have size at most $\frac{2}{3}n$

so we get the recurrence

$$T(n) \leq T(2n/3) + \Theta(1)$$

which, by case 2 of the master theorem, has solution

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

(Or, MAXHEAPIFY takes $O(h)$ time when node $A[i]$ has height h in the heap)

BuildMaxHeap(A): Building a Heap

Intuition: use MAXHEAPIFY in a bottom-up manner to convert A into a heap

- leaves are already heaps
- start at parents of leaves
- then, grandparents of leaves
- etc.

BUILDMAXHEAP(A)

 heapsize(A) := length(A)

for $i := \text{length}(A)/2$ **downto** 1

 MAXHEAPIFY(A, i)

Running Time of BUILDMAXHEAP

- about $n/2$ calls to MAXHEAPIFY ($O(n)$ of them)
- each one takes $O(\log n)$ time
- $\implies O(n \log n)$ time total

Actually, can show BUILDMAXHEAP runs in $O(n)$ time, but won't affect the running time of the algorithm

$$\sum_{h=0}^{\log n} (\# \text{nodes height } h) O(h) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right) = O(n)$$

Heaps: Space, Inserting, . . .

Space Usage for Heapsort

- An array implementation of a heap used $O(n)$ space
 - one array element for each node in heap
- Heapsort uses $O(n)$ space – and is **in place**

Inserting an element into a heap

- increment heapsize and “add” new element (key) to the end of the array
- walk-up tree from (new) leaf to root, swapping key with parent’s key until find a parent larger than the key

MAX-HEAPINSERT(A, key)

 heapsize(A) := heapsize(A) + 1

i := heapsize(A)

while $i > 1$ and $A[\text{parent}(i)] < key$

$A[i] := A[\text{parent}(i)]$

$i := \text{parent}(i)$

$A[i] := key$

MAX-HEAPINSERT Running Time: $O(\log n)$

– time to traverse leaf to root path (height = $O(\log n)$)

QuickSort

```

QUICKSORT( $A, p, r$ )
  if  $p < r$ 
    select  $e$  an index in  $[p, r]$  /**pick pivot element**/
    swap(  $A[e], A[r]$  )
     $q :=$  PARTITION( $A, p, r$ ) /**break up A wrt pivot element**/
    QUICKSORT( $A, p, q - 1$ )
    QUICKSORT( $A, q + 1, r$ )

```

Notes:

1. There are many ways to select the pivot (partition) element
 - always pick $A[p]$ or $A[r]$ (i.e., $i = p$ or $i = r$)
 - pick an element of $A[p, r]$ at random
 - pick several elements of $A[p, r]$ and pick the median of those
 - etc.
2. PARTITION(A, p, r) rearranges A such that

$$\begin{array}{ll}
 A[i] \leq \text{pivot element} & \text{for } p \leq i \leq q \\
 A[i] \geq \text{pivot element} & \text{for } q < i \leq r
 \end{array}$$

Quicksort is a simple divide-and-conquer algorithm:

- **divide step:** pick the pivot element and PARTITION(A, p, r)
- **conquer step:** recursively sort two subarrays
- **combine step:** do nothing, quicksort is an *in place* algorithm

Quicksort: partitioning the array

```

PARTITION( $A, p, r$ )
   $x := A[r]$  /** pivot is last element in  $A[p, r]$  **/
   $i := p - 1$ 
  for  $j := p$  to  $r-1$ 
    do if  $A[j] \leq x$ 
      then  $i := i + 1$ 
         exchange ( $A[i], A[j]$ )
  exchange ( $A[i + 1], A[r]$ ) /**swap pivot so in middle**/
  return ( $i + 1$ ) /**return final index of pivot**/

```

Correctness (Invariants): As $\text{PARTITION}(A, p, r)$ executes, the array $A[p, r]$ is ‘partitioned’ into four (possibly empty) regions. At the start of each iteration of the **for** loop, the elements in each region satisfy certain properties with respect to the pivot element x . These can be used to form a *loop invariant*.

R1: $A[p, i]$ these elements here are \leq the pivot x

R2: $A[i + 1, j - 1]$ these elements here $>$ than the pivot x

R3: $A[j, r - 1]$ these elements here have no relation to pivot x

R4: $A[r] = x$ this element is the pivot element x

Running Time = $O(n)$

- In each iteration of the **for** loop increases j , and there are $(r - 1) - p$ iterations. So, PARTITION runs in time linear in the size of A .

Quicksort Running Time

What is the running time of Quicksort?

It is a divide-and-conquer algorithm, so:

$$T(n) = T(q - p) + T(r - q) + O(r - p)$$

what this is depends on....

- the position of q in the array $A[p, r]$
- unfortunately, we don't know this....

So what do we do?

We can analyze

- worst-case
- best-case
- average-case

Quicksort: worst-case running time

What is the worst-case for Quicksort?

Intuition: when each partition results in a split which has $n - 1$ elements in one subproblem, and zero in the other (the pivot is always in ‘correct’ spot and is not included in any subproblem).

$$\begin{aligned}
 T(n) &= T(0) + T(n - 1) + \Theta(n) \\
 &= T(n - 1) + \Theta(n) \\
 &= [T(n - 2) + \Theta(n - 1)] + \Theta(n) \\
 &= [T(n - 3) + \Theta(n - 2)] + \Theta(n - 1) + \Theta(n) \\
 &\vdots \\
 &= \sum_{k=1}^{k=n} \Theta(k) \\
 &= \Theta\left(\sum_{k=1}^{k=n} k\right) \\
 &= \Theta(n^2)
 \end{aligned}$$

... so, there is *at least one* execution that takes **at least** n^2 time

Quicksort: worst-case running time

but is this the worst???? yes!

The “real” upperbound is expressed by:

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n - q - 1) + \Theta(n))$$

we deal with this by ‘guessing’ $T(n) \leq cn^2$ and verifying it inductively

$$\begin{aligned} T(n) &= \max_{1 \leq q \leq n-1} (cq^2 + c(n - q)^2 + \Theta(n)) \\ &= c \max_{1 \leq q \leq n-1} (q^2 + (n - q)^2) + \Theta(n) \end{aligned}$$

we can now use calculus to see that $q^2 + (n - q)^2$ is maximum at its endpoints (since the second derivative is positive),

that is, when $q = 1$ or $q = n - q - 1$ (which is what our intuition told us)

Quicksort: best-case running time

What is the best-case for Quicksort?

Intuition: perfectly balanced splits – each partition gives an almost equal split ($\lfloor n/2 \rfloor$ elements in one, $\lceil n/2 \rceil - 1$ elements in the other, and the pivot).

$$\begin{aligned} T(n) &\leq T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n) \end{aligned}$$

using Master Theorem:

- $a = 2, b = 2, f(n) = n$
- $\log_b a = 1$
- $n^{\log_b a} = n^1 = n$
- compare $f(n)$ and $n^{\log_b a}$: $n = \theta(n)$, or $f(n) = \Theta(n^{\log_b a})$
- so we're in case 2 of Master Theorem, and

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

... so, there is *at least one* execution that takes **at most** $n \log n$ time

Quicksort: average-case running time

What is the average-case for Quicksort?

Intuition: some splits will be close to balanced and others will be close to imbalanced... \implies good and bad splits will be **randomly** distributed in recursion tree...

Important: we'll only have a (asymptotically) bad running time if we have many bad splits *in a row*

- a bad split followed by a good split results in a good partitioning after one extra step!
- implies we'll still get $\Theta(n \log n)$ running time! (albeit with a larger constant)

How to modify Quicksort to get good average case behavior on **all** inputs??

Answer: **Randomization!**

- randomly permute input
 - have a tree of possible executions, and most of them finish fast
- choose partitioning element e randomly at each iteration
 - easier to analyze, same good behavior

Quicksort: A randomized version

Problem: We've made an assumption that all permutations of the input numbers are equally likely. In many common situations, however, this does not hold. One way we can attempt make it hold is by using randomization to obtain good average case behavior over all inputs.

```
RANDOMIZED-PARTITION( $A, p, r$ )  
   $i :=$  random number in  $[p, r]$   
  exchange ( $A[r], A[i]$ )  
  return PARTITION( $A, p, r$ )
```

And the randomized QUICKSORT will call RANDOMIZED-PARTITION instead of PARTITION.

Its expected running time is $O(n \log n)$.

Lowerbounds for Sorting

How fast can we sort???

We have seen several algorithms that run in $\Omega(n \log n)$ time in the worst-case (there is some input on which the algorithms run in at least $\Omega(n \log n)$ time):

- mergesort
- heapsort
- quicksort

All the above algorithms are **comparison based**: *the sorted order they determine is based only on comparisons between the input elements.*

Can we do better??? Unfortunately, no!!

Any comparison based sort must make $\Omega(n \log n)$ comparisons in the worst-case to sort a sequence of n elements.

But... How can we prove this?

We'll use the **decision tree model** to represent *any* sorting algorithm, and then argue that no matter the algorithm, there is **some** input which will cause it to run in $\Omega(n \log n)$ time.

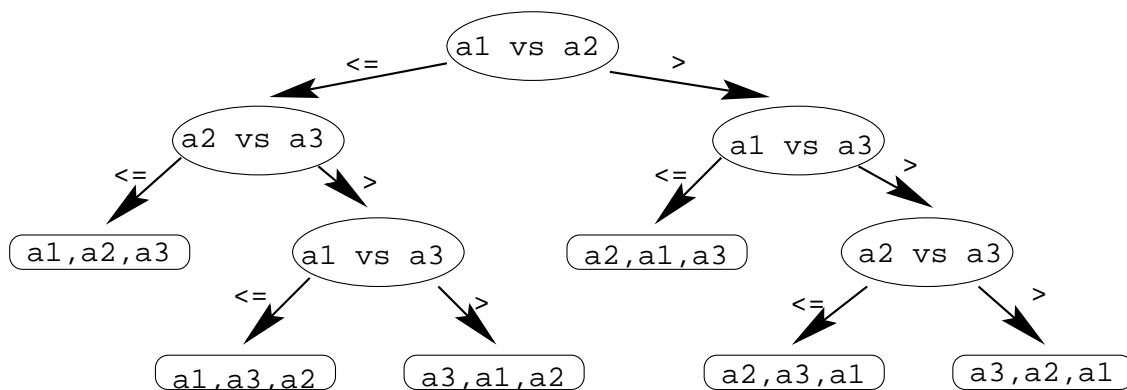
The Decision Tree Model

Given any **comparison-based** sorting algorithm, we can represent its behavior on an input of a *certain size* (i.e., a particular value of n) by a **decision tree**:

- **only** consider the comparisons in the algorithm (the other operations only make the algorithm take longer)
- each internal node in the decision tree corresponds to one of the comparisons in the algorithm
- start at root and do first comparison: if \leq take left branch, if $>$ take right branch
- etc.
- each leaf represents one (correct) ordering of the input
 - one leaf for each of $n!$ possible orders

\implies One decision tree for each algorithm and input size

Example: Insertion sort with $n = 3$ ($3! = 6$ leaves)



The $\Omega(n \log n)$ Lowerbound

Key Points regarding Decision Tree:

- There must be $n!$ leaves, one for each possible ordering of n elements (permutations)
- Length (#edges) of longest path in decision tree (its height)
= worst-case number of comparisons
 \leq worst-case number of operations of algorithm (lowerbound on time)

Theorem: *Any decision tree for sorting n elements has height (length of longest path) $\Omega(n \log n)$*

Proof:

Let h be the height of the tree (length of longest path).

- tree has $\geq n!$ leaves
- tree is binary so has $\leq 2^h$ leaves

$$\begin{aligned} 2^h &\geq n! \\ h &\geq \log_2(n!) \\ &= \Omega(n \log n) \quad \text{by Stirling's approx } n! > \left(\frac{n}{e}\right)^n \end{aligned}$$

□

\implies any comparison-based algorithm takes $\Omega(n \log n)$ time **in the worst-case**.

Beating the lowerbound... non-comparison based sorts

Idea: Algorithms that **are not** comparison-based might be faster...

There are three such algorithms in Chapter 8:

- counting sort
- radix sort
- bucket sort

These algorithms:

- run in $O(n)$ time (under certain conditions)
- use information about the values to be sorted (counting, bucket sort)
- or operate on 'pieces' of the input elements (radix sort)

Counting Sort

Requirement: input elements integers in known range (e.g., $[1, k]$ for constant k)

Idea: for each input element x , find the number of elements $\leq x$ (say m) and put x in the $(m + 1)$ st spot in the output array

COUNTING SORT(A, k)

arrays $A[1, n]$, $A_{out}[1, n]$, $C[1, k]$

initialize $C[i] = 0$ for $i = 1, k$

for $i := 1, n$

$C[A[i]] := C[A[i]] + 1$

 /** now C[j] holds # elements equal to j **/

for $i := 1, k$

$C[i] := C[1] + C[2] + \dots + C[i]$

 /** now C[j] holds # elements $\leq j$ **/

for $i := 1, n$

 index := $C[A[i]]$ /** where to put A[i] **/

$A_{out}[\text{index}] := A[i]$ /** put A[i] there **/

$C[A[i]] := C[A[i]] - 1$ /** decrement count C[A[i]] **/

Running Time: $O(k + n)$

- two **for** loops of size k
- two **for** loops of size n

When does Counting sort beat an $\Omega(n \log n)$ sort?

for $k = o(n \log n)$ (can get close to $O(n)$ for larger k using hash tables)

Radix Sort

Let d be the number of **digits** in each input number.

RADIX SORT(A, d)
for $i := 1, d$
stably sort A on digit i

Definition: A sorting algorithm is **stable**, that is, the relative order of elements with the same key is the same in the output array as in the input array (counting sort is stable)

Note:

- radix sort sorts the least significant digit first!
- correctness can be shown by induction on the digit being sorted
- often counting sort is used as the internal sort
- can make fewer than d passes by considering more than one digit at a time

Running Time: $O(dT_{ss}(n))$

- T_{ss} is the time for the internal sort
- counting sort gives $T_{ss}(n) = O(k + n)$
 \implies so $O(dT_{ss}(n)) = O(d(k + n))$
 \implies which is $O(n)$ if $d = O(1)$ and $k = O(n)$
- if $d = O(\log n)$ and $k = 2$ (common for computers),
then $O(d(k + n)) = O(n \log n)$

Bucket Sort

Assumption: input elements distributed uniformly over some known range, e.g., $[0, 1)$ (Section 6.2 in text defines uniform distribution)

BUCKET SORT(A, x, y)

1. divide interval $[x, y)$ into n equal-sized subintervals (**buckets**)
2. distribute the n input keys into the buckets
3. sort the numbers in each bucket (e.g., with insertion sort)
4. scan the (sorted) buckets in order and produce output array

Running Time:

Step 1: $O(1)$ for each interval $\implies O(n)$ time total

Step 2: $O(n)$ time

Step 3: *What is the expected number of elements in each bucket?*

- since the elements are uniformly distributed over $[x, y)$, we expect very few elements in each bucket
- intuitively, we expect $O(1)$ elements in each bucket, so all the insertion sorts of the buckets will take $O(n)$ time total
- (a formal argument is contained in the text)

Step 4: $O(n)$ time to scan the n buckets containing a total of n input elements

\implies expected total running time of bucket sort is $O(n)$

Order Statistics

Let A be an ordered set containing n elements

Definition: The i th order statistic is the i th smallest element

- minimum = 1st order statistic
- maximum = n th order statistic
- median = $\lfloor \frac{n+1}{2} \rfloor = \lceil \frac{n+1}{2} \rceil$
- etc.

The Selection Problem: Find the i th order statistic for a given i

input: A set A of n (distinct) numbers, and
a number i , $1 \leq i \leq n$

output: The element $x \in A$ that is larger than exactly $(i - 1)$ elements of A

```

NAIVESELECTION( $A, i$ ) {
     $A' := \text{FAVORITESORT}(A)$ 
    return  $A'[i]$ 
}

```

Running Time: $O(n \log n)$ for comparison based sorting

Can we do better????

Finding Minimum (or Maximum)

```
MINIMUM( $A$ ) {  
    min :=  $A[1]$   
    for  $i := 2$  to  $n$   
        min := min(min,  $A[i]$ )  
}
```

Running Time: $O(n)$

- just scan input array
- **exactly** $n - 1$ comparisons

Is this the best possible? **Yes!!**

Why are $n - 1$ comparisons necessary? (lowerbound)

- Any algorithm that finds the minimum must compare all elements with the 'winner' (think of a tournament)
- so... there must be at least $n - 1$ losers (and each loss requires a comparison)
- Another way – we must 'look' at every key, otherwise the missed one may be the minimum. Each look requires a comparison (except the first).

Selection in linear expected time

```

RANDOMIZED-SELECT( $A, p, r, i$ ) {
  if  $p = r$  then return  $A[p]$ 
   $q :=$  RANDOMIZED-PARTITION( $A, p, r$ )
  if ( $i$  is in  $[p, q]$ )
    then return RANDOMIZED-SELECT( $A, p, q, i$ )
    else return RANDOMIZED-SELECT( $A, q, r, i - (q - p + 1)$ )
}

```

Note:

- RANDOMIZED-PARTITION first swaps $A[p]$ with a random element of A and then proceeds as in regular PARTITION.
- RANDOMIZED-SELECT is like RANDOMIZED QUICKSORT – except we only need to make *one* of the recursive calls

Running Time:

- worst-case – unlucky with bad $1 : n - 1$ partitions
 $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$
- best-case – really lucky and quickly reduce subarrays
 $T(n) = \Theta(n)$
- average-case – ??
 – like Quicksort, will be asymptotically close to best case

Selection in linear expected time – average case

$$T(n) \leq O(n) + \sum_x x \cdot \Pr[\text{max subproblem size} = x] \quad (1)$$

$$\leq O(n) + \sum_{k=1}^{n-1} T(\max(k, n-k)) \cdot \frac{1}{n} \quad (2)$$

$$\leq O(n) + \left(T(n-1) + 2 \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) \right) \cdot \frac{1}{n} \quad (3)$$

$$= O(n) + \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) \quad (4)$$

$$= O(n) \quad (5)$$

Notes:

- we 'get rid' of $\frac{1}{n}T(n-1)$ from line (3) to line (4) since $\frac{1}{n}T(n-1) \leq \frac{1}{n}O(n^2) = O(n)$
(we fold it into the existing $O(n)$ term)
- the bound in line (5) is shown by the 'guess and verify' method (substitution)
– guess $T(n) \leq cn$ and verify by induction

Selection in linear worst-case time

Key: Guarantee a 'good' split when array is partitioned – will yield a method that *always* runs in linear time

SELECT(A, p, r, i)

1. divide input array A into $\lfloor \frac{n}{5} \rfloor$ groups of size 5 (and one leftover group of size < 5)
2. find the median of each group of size 5
3. call **SELECT** recursively to find x , the median of the $\lceil \frac{n}{5} \rceil$ medians
4. partition array around x , splitting it into two arrays of $A[p, q]$ and $A[q+1, r]$
5. if $(i \leq k)$ **/**like Randomized-Select**/**
 then call **SELECT**($A[p, q], i$)
 else call **SELECT**($A[q+1, r], i - (q - p + 1)$)

Main idea: this guarantees that x causes a 'good split' (at least a constant fraction of the n elements is $\leq x$ and constant fraction is $> x$)

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

elements are $> x$ (or $< x$)

\implies worst-case split has $\frac{7}{10}n + 6$ elements in 'big' problem

Selection in worst-case linear time

Running Time:

1. $O(n)$ (break into groups)
2. $O(n)$ (finding median of 5 numbers is constant time)
3. $T(\lceil \frac{n}{5} \rceil)$ (recursive call to find median of medians)
4. $O(n)$
5. $T(\frac{7}{10}n + 6)$

$$T(n) = T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n + 6) + O(n)$$

We'll solve this using the substitution method (guess and verify)
 guess $T(n) \leq cn$

$$\begin{aligned} T(n) &= T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n + 6) + O(n) \\ &\leq c\lceil \frac{n}{5} \rceil + c(\frac{7}{10}n + 6) + O(n) \\ &\leq c(\frac{n}{5} + 1) + \frac{7}{10}cn + 6c + O(n) \\ &= cn - (\frac{1}{10}cn - 7c) + O(n) \\ &\leq cn \end{aligned}$$

When does the last step hold?

- $n \geq 80$ implies $(\frac{1}{10}cn - 7c)$ is positive
- c big enough makes $O(n) - (\frac{1}{10}cn - 7c)$ negative, so that cn is larger than second to last line