The complexity of Sorting and sorting in linear-time

Median and Order Statistics

Chapter 8 and Chapter 9

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

The sorting algorithms we have examined so far are **comparison** and **exchange** based. This means that in order to sort keys, a pair of input keys are compared and depending on the outcome of the comparison they may or may not be exchanged.

In this section we prove that no matter what algorithm is to be used for comparison and exchange sorting, the worst-case running time of the algorithm will be $\Omega(n \lg n)$ **for some input instance**. In order to prove this bound we shall introduce a tree called the decision tree.

In the remainder of this subject we shall present two more sorting algorithms whose main feature is that they do not use comparisons and exchanges for sorting. They inspect instead the values of the keys and the way they work is based on some previously acquired knowledge on the values of the keys to be sorted. The three algorithms are listed below.

Count Sort/Bin Sort It assumes that keys are from an integer range $1, \ldots, k$. It sorts n keys in that range in time $O(n+k)$. This algorithm is also known as **Bin-Sort**.

Radix Sort It assumes that keys are numbers/strings of some length whose digits/characters are from an integer range $1 \ldots k$. It sorts n d-digit keys in time $O(d(n+k))$.

Bucket-Sort A third algorithm that will not be covered in detail, assumes than n keys are uniformly distributed in a range $[a, b]$. It sorts the keys in **expected time** $O(n)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

In a comparison based sorting algorithm we compare elements of the input using operations like $<, >, =, \geq, \leq$ to determine the relative order of the input keys. Without loss of generality we may assume that the keys are distinct (Note: To make keys distinct if they are not we can create an extended key. This way from key k we create key (k, i) where i is the index of key k in the input sequence). Therefore $=$ tests can be avoided, and the remaining operations are equivalent ie. we only need $\leq$ based comparisons to draw conclusions.

The decision tree model

For an input of given size n, a **decision tree** is a full binary tree that represents the comparisons between keys performed by a particular sorting algorithm; control, data exchanges and other details of the algorithm are ignored. The decision tree is thus a binary tree whose internal nodes are pairs $a_i : a_j$ for some $i, j \leq n$. Each leaf is a permutation of the input indices (not keys) $\langle p(1), p(2), \ldots, p(n) \rangle$.

The execution of the sorting algorithm on some input of size n is a path from the root of the decision tree to a leaf. The path shows the comparisons that were performed between any two keys.

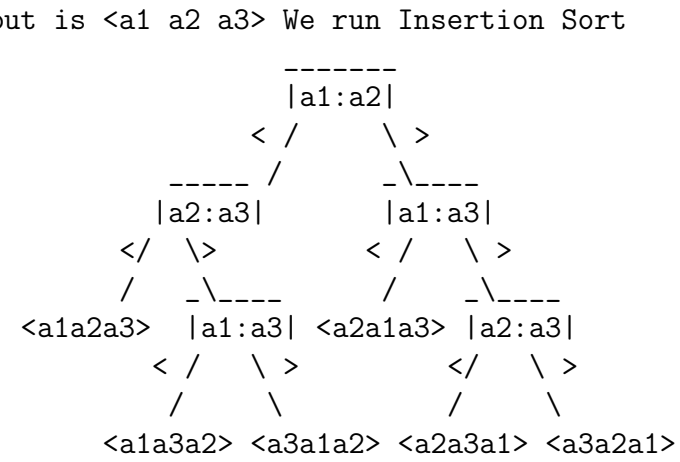
If at some node a left path was taken this means that $a_i \leq a_j$ was decided, otherwise $a_i > a_j$.

When a leaf is reached, this means that the output sequence has been decided and the sorting algorithm has established say the ordering $\langle a_{p(1)}, a_{p(2)}, \ldots, a_{p(n)} \rangle$ for the input (we reestablish the output sequence from the output permutation).

As the number of permutations on n inputs is $n!$ and each permutation must appear as a leaf (provided that the sorting algorithm is correct i.e. it DOES sort) of the decision tree we conclude that any decision tree for sorting SHOULD have at least $n!$ leaves.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A three-key decision tree for InsSort



All 3!=6 permutations of the input sequence appear as leaves.
If <a1 a2 a3> is the reached leaf it means that in the input
<a1 a2 a3> we have that a1 < a2 < a3
If however <a3 a2 a1> is the reached leaf it means that in the
input a3 < a2 < a1

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Worst Case bound

As a path from the root to a leaf gives a computation trace of the algorithm for some input, the longest path represents the worst case number of comparison steps required to sort an input sequence by that number of steps. (In addition the path provides us with the worst-case input as well).

Therefore the worst case bound for a comparison based algorithm is the height of its decision tree. We are going to establish a lower bound on the height of decision trees of at least $n!$ leaves to derive a lower bound for comparison based sorting algorithms.

Theorem. Any decision tree that sorts n keys (using comparisons) has height $\Omega(n \lg n)$.

Proof. A decision tree that sorts n keys must have at least $n!$ leaves, since n (distinct) keys generate $n!$ permutations representing a distinct sorted sequence. A binary tree of height h has at most 2^h leaves. Therefore we need that $2^h \geq n!$ ie $h \geq \lg(n!)$. From Stirling's formula for the factorial (2.11), we get $n! \geq (n/e)^n$. This implies that $h \geq n \lg n - n \lg e = \Omega(n \lg n)$.

Corollary. Heapsort and mergesort are asymptotically optimal.

(c) Copyright A. Gerbessiotis. All rights reserved.

Let us assume that n keys to be sorted take values from a set $0, \ldots, k-1$. In many case we shall assume that $k = O(n)$. We are going to present an algorithm whose running time is $O(n+k)$. Then for $k = O(n)$ this is $O(n)$, ie a linear time algorithm for sorting exists.

The main idea for count sort is that since the size of the set of values of a key is "small" we are going to count the number of time each key value appears in the sorted sequence. This way we can then retrieve the number of keys whose value is less than some value x, the we know the position for x in the output sequence. For example if number of keys less than x is 20, the position of x is 21 if there are no duplicates. If however we do have 4 keys with values x, then we know their positions are $21, 22, 23, 24$ in the output sequence. Three arrays are to be used $I[1..n]$ that holds the input, $O[1..n]$ that hold the output and $C[0..k-1]$ that counts the number of keys with values $0 \leq j < k$.

```
 Count-Sort(I,O,n,k)
   // Initialize Counter array C
1. for(i=0;i<k;i++)
2.    C[i]=0;
  // If key I[j] is  m increment C[m] by one
3. for(j=1;j<=n;j++)
4.    C[I[j]] ++;
  // C[j] : number of keys with value j
  // Make C[i] to hold number of keys with values at most i.
5. for(i=1;i<k;i++)
6.   C[i] = C[i]+C[i-1];
7. for(j=n;j>=1;j--) {
8.    O[C[I[j]]]=I[j];
9.    C[I[j]]--;
   }
```

(c) Copyright A. Gerbessiotis.

Analysis Steps 1 and 2 take $O(k)$ time. Steps 3 and 4 take $O(n)$ time. Steps 5 and 6 $O(k)$ time and 7,8 and 9 $O(n)$ time. Total running time is $O(n+k)$.

Note that the previous lower bound fails to hold as count sort inspects values of keys and **DOES NOT USE** comparisons for sorting.

Note also that it is a **stable** algorithm, ie equal keys in the input sequence preserve their relative order in the output sequence.

Generalizing CountSort: RadixSort

Suppose that the range of keys to be sorted is $0, \ldots k^d - 1$ for some d. One way to solve this sorting problem is to use countsort and solve it in time $O(n + k^d)$. The term k^d, however may be too large. We can write down these values in radix-d notation as d digits each one holding k different values. $d = 32$, $k = 2$ is a frequent representation of integers on micropocessors. We shall show how to utilize d phases of Countsort, each one taking $O(n+k)$ time to solve this problem in $O(d(n+k))$ time.

```
Left To Right Operation (DOES NOT SORT)
    V            V           V            <<<<<<< Shows which column is
    8 1 2      1 2 4     8 1 2     8 1 2          used as key for countsort
    2 9 5  --> 2 9 5 --> 1 2 4 --> 1 2 4
    1 2 4      8 1 2     2 9 5     2 9 5
Right To Left Operation (IT SORTS)
        V       V        V
    8 1 2     8 1 2      8 1 2     1 2 4
    2 9 5 --> 1 2 4  --> 1 2 4 --> 2 9 5
    1 2 4     2 9 5      2 9 5     8 1 2
```

The Algorithm

One may think that starting the sorting from the most significant position is a way to go. We are going to perform d iteration of the following loop. We start from the l-th most significant position and sort the keys according to this position using count sort into k subsequences (one for each value of a digit). Total time is $O(n+k)$. We then sort with the $l+1$ most significant digit the obtained sequence. To retain the previous order, we must perform count sort on each of the k subsequences separately which would require just for initialization of counters $O(k^2)$ and scanning of input keys $O(n)$ for a total $O(k^2+n)$. When the least significant bit is reached, the time to sort the n keys according to that digit will be $O(n+k^d)$. The reason for the failure of this method is that past ordering is not used in the sorting in later phases.

We peform count sort in d phases but we start from the least significant bit position. Each time we move a digit to the left, we split the whole set of currently ordered sets into k subsequences as required (and not each subsequence, as before). The major observation is that keys sorted with respect to least significant bit position $l+1$ are also sorted if we restrict the values of each key to the value implied by position $l+1\ldots 1$ and ignore most significant digits. For this to hold, it is essential that the count sort operation is stable. If two inputs are $21, 23$ and are sorted on the second lsb (least significant) digit (i.e. most significant digit in this case) we want the relative order to be maintained i.e. 21 appears before 23. (Note that $C[2]=2$ and $C[i]=0$ for all $i\neq 2$). A non-stable count sort on digit 2 may yield $23, 21$ which is not what we wanted. The Radix Sort algorithm is quite simple

```
Radix-Sort(A,d,n)
// digit 1 least significant , d most significant
1. for(i=1;i<=d;i++)
2.  Countsort A on digit i
```

Analysis. The algorithm has d iterations each one requiring $O(n+k)$ steps (line 2) ie a total $O(d(n+k))$ (for lines 1-2).

If numbers are 32-bit we may perform 32 radix-2 rounds of countsort on 0-1 values, or 4 radix-256 rounds of countsort or a single radix-2^{31} round of countsort to sort the numbers.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A. Bits and Values

Fact 0. The integer numbers $0, \ldots, m-1$ can be represented by $\lceil \lg m \rceil$ bits.
Fact 1. With x bits we can represent 2^x distinct non-negative integer values: $0, 1, \ldots, 2^x - 1$.
Fact 2. Therefore, with $\lg y$ bits we can represent $2^{\lg y} = y$ values $0, \ldots, y-1$.
Fact 3. What is 110?

- Base 2 (radix-2, binary) 110 is $110 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6$.
- Base 10 (radix-10,decimanl) 110 is $110 = 1 \cdot 10^2 + 1 \cdot 10^1 + 0 \cdot 10^0 = 110$.
- Base 16 (radix-16,hexadecimal) 110 is $110 = 1 \cdot 16^2 + 1 \cdot 16^1 + 0 \cdot 16^0 = 272$.

Fact 4. Countsort sorts n keys in the range $0, \ldots, k-1$ in time $O(n+k)$.
Fact 5. Count-Sort sorts n keys in the range $0, \ldots, k^d - 1$ in time $O(n + k^d)$.
Fact 6. Radix-Sort sorts n keys in the range $0, \ldots, k^d - 1$ in time $O(d(n+k))$.
Fact 7. Suppose we have n keys that 32-bit integers. Radix-Sort requires:

- 1-digit keys: View each key as an 1-digit radix-2^{32} integer i.e. $k = 2^{32}$, $d = 1$. Running time is $O(1 \cdot (n + 2^{32}))$.
- 2-digit keys: View each key as an 2-digit radix-2^{16} integer i.e. $k = 2^{16}$, $d = 2$. Running time is $O(2 \cdot (n + 2^{16}))$.
- 4-digit keys: View each key as an 4-digit radix-2^8 integer i.e. $k = 2^8$, $d = 4$. Running time is $O(4 \cdot (n + 2^8))$.
- 32-digit keys: View each key as an 32-digit radix-2 integer i.e. $k = 2$, $d = 32$. Running time is $O(32 \cdot (n + 2^1))$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Fact 8. We have n keys in the range $0, \ldots, n^2 - 1$. How dow we sort them?

Proof (Fact 8): If we use Merge-Sort/Heap-Sort running time is $O(n \lg n)$. If we use count-sort for $n = n$ and $k = n^2$ running time is $O(n + n^2) = O(n^2)$.

However Radix-Sort helps. Each integer in the range $0, \ldots, n^2 - 1$ is represented by (Fact 0) about $\lg n^2 = 2 \lg n$ bits. Group the bits into two groups of $\lg n$. Each group becomes a digit (i.e. we have 2 groups). Each group of $\lg n$ bits takes (Fact 2) $2^{\lg n} = n$ distinct values i.e. it can be viewed as a digit that takes that many values, i.e. it becomes a radix-n digit. We have $d = 2$ digits each one radix-n i.e. $k = n$. By Radix-Sort running time is $O(d(n + k)) = O(2(n + n)) = O(n)$. Radix-Sort beats Merge-Sort/Heap-Sort and Count-Sort. □

Fact 9. We have n keys in the range $0, \ldots, 2^n - 1$. How dow we sort them?

Proof (Fact 9): If we use Merge-Sort/Heap-Sort running time is $O(n \lg n)$. If we use count-sort for $n = n$ and $k = 2^n$ running time is $O(n + 2^n) = O(2^n)$.

However Radix-Sort helps. Each integer in the range $0, \ldots, 2^n - 1$ is represented by (Fact 0) about $\lg 2^n = n$ bits. Group the bits into two groups of $\lg n$ bits. Each group becomes a digit (i.e. we have $n/\lg n$ groups). Each group of $\lg n$ bits takes (Fact 2) $2^{\lg n} = n$ distinct values i.e. it can be viewed as a digit that takes that many values, i.e. it becomes a radix-n digit. We have $d = n/\lg n$ digits each one radix-n i.e. $k = n$. By Radix-Sort running time is $O(\frac{n}{\lg n}(n + k)) = O(\frac{n}{\lg n}(n + n)) = O(n^2/\lg n)$. Radix-Sort/Count-Sort are beaten by Merge-Sort/Heap-Sort, though they still beat in the worst case Insertion-Sort/Quick-Sort etc. □

You are advised to revise sections 6.1, 6.2 and 6.3 if you are not current with some of the following concept.

Problem. n balls are thrown independently into n bins. Each ball has the same probability entering any of the n bins. Let n_i be the number of balls into the i-th bin. Find $E[n_i^2]$ (ie the expected value of n_i^2).

Solution.

We introduce some notations first

1. $n_i, 1 \leq i \leq n$ is the number of balls falling into the i-th bin.

2. $Y_{ij}, 1 \leq i, j \leq n$ is a variable that take two values. $Y_{ij} = 1$ if the j-th balls falls into the i-th bin. If this is not the case, $Y_{ij} = 0$.

3. Because of 1. and 2. we have that $n_i = \sum_j Y_{ij}$

4. Ball j falls into bin i with probability $1/n$. Therefore $Prob(Y_{ij} = 1) = 1/n$ and thus $Prob(Y_{ij} = 0) = 1 - 1/n$.

5. $E[Y_{ij}] = 1Prob(Y_{ij} = 1) + 0Prob(Y_{ij} = 0) = 1(1/n) + 0(1 - 1/n) = 1/n$ because of 2.

6. $E[Y_{ij}^2] = 1^2Prob(Y_{ij} = 1) + 0^2Prob(Y_{ij} = 0) = 1(1/n) + 0(1 - 1/n) = 1/n$ because of 2.

7. $var(Y_{ij}) = E[Y_{ij}^2] - E^2[Y_{ij}] = 1/n - 1/n^2 = (1/n)(1 - 1/n)$. This follows from the definition of variance.

8. $E[n_i] = E[\sum_j Y_{ij}] = \sum_{j=1}^{n} E[Y_{ij}] = n(1/n) = 1$ by 3. and the fact that the expectation of the sum is the sum of the expectations.

9. $var[n_i] = var[\sum_j Y_{ij}] = \sum_j var[Y_{ij}] = n(1/n)(1 - 1/n) = 1 - 1/n$ by the definition of variance and the fact that the variance of the sum is the sum of variances for **INDEPENDENT RANDOM VARIABLES**. We note that the throwing of ball j is independent of the throwing of ball k, $k \neq j$.

10. $var(n_i) = E[n_i^2] - E^2[n_i]$. Therefore by way of 9., 8., we get that $E[n_i^2] = var(n_i) + E^2[n_i] = (1 - 1/n) + 1 = 2 - 1/n$.

(c) Copyright A. Gerbessiotis.

Bucket Sort is an average case algorithm. The assumption is that the input keys are in the range $[a, b]$. We can alway transform the keys to a range $[0, 1]$ by shifting and translation operations (ie. $x \in [a, b]$ means that $(x - a)/(b - a) \in [0, 1]$.

The idea is to divide the interval into n equally spaced subintervals that will be called **buckets**. Let us index buckets $0, \ldots n-1$ left to right. We then distribute each key to its destination bucket. We sort the keys per bucket using our favorite comparison based algorithm. For two buckets $i < j$ the elements of i are smaller than any element of j. Therefore after the comparison sort all keys are sorted. The algorithm is shown below. Input is $A[0..n-1]$ and buckets are represented by linked lists with heads in array $B[0..n-1]$.

```
   Bucket-Sort(A,n)
1. for(i=0;i<n;i++)
2.   A[i] belongs to bucket B[floor(n A[i])]
3. for(i=0;i<n;i++)
4.   sort keys in bucket i ie elements of list B[i] with say insertion sort
//   Note that the key are the actual values in A[..]
5. Concatenate list B[0], B[1], B[2], ...
```

The key observation to the proof of correctness is that $A[i] \leq A[j]$ implies $\text{floor}(nA[i]) \leq \text{floor}(nA[j])$. Let n_i be the number of keys in bucket i (ie in list $B[i]$). Each key as it is drawn uniformly at random has probability $p_i = 1/n$ to fall into $B[i]$. Then the expectation of the size of n_i is $E[n_i] = 1$ and $E[n_i^2] = 2 - 1/n$. Step 4 requires for bucket i $O(n_i^2)$ steps if insertion sort is used. Therefore the time for the loop of step 3-4 is

$$T(n) = \sum_{i=0}^{n-1} O(n_i^2) = O(\sum_{i=0}^{n-1} n_i^2)$$

Then, by taking expectations on both sides we get

$$E[T(n)] = E[\sum_{i=0}^{n-1} O(n_i^2)] = E[O(\sum_{i=0}^{n-1} n_i^2)] = O(\sum_{i=0}^{n-1} E[n_i^2]) = O(\sum_{i=0}^{n-1}(2 - 1/n)) = O(n(2 - 1/n)) = O(2n - 1) = O(n).$$

When an input sequence of keys is sorted, as a byproduct of sorting we also know what the maximum and minimum keys of the sorted sequence are. They are elements $A[1]$ (minimum) and $A[n]$ (maximum) of the sorted output array $A[1..n]$.

We also know the i-th smallest key: it is $A[i]$. This is also the $n-i$ largest key (the 1-st smallest is the minimum and 1-st largest is the maximum).

The i-th smallest of a set of keys is also known as the **i-th order statistic**.

One of the interesting statistics is what is called the **median**, The **lower median or median** is the $\lfloor \frac{n+1}{2} \rfloor$ statistic. The **upper median** is the $\lceil \frac{n+1}{2} \rceil$ statistic.

The **selection problem** is defined as follows.
Input. A set S of n (distinct) keys and a number i such that $1 \leq i \leq n$.
Output The element $s \in S$ such that s is larger than $i-1$ other elements of S.

One way to solve the Selection problem is by sorting and then reading off the i-th sorted element (from the left, where leftmost is the smallest of the n elements).

(c) Copyright A. Gerbessiotis. All rights reserved.

Problem 1. Find the minimum of n keys?

Answer: An elementary upper bound on the number of comparisons is $n-1$. This is also a lower bound as all the other keys need to be read and ruled out before a conclusion on the minimum can be drawn. This can also be shown by a tournament argument where every key but the winner must lose a match (i.e. a comparison must result in this element not being the minimum).

Problem 2. Solve the MINMAX problem i.e. find both the minimum and the maximum at the same time.

Answer: A relevant question is how long it takes to find **both** the minimum and the maximum. An easy upper bound is $2n-2$ comparisons by using the previous algorithm twice. Under this algorithm every key is compared twice, once with the current minimum and once with the current maximum.

A better algorithm is the following

> Take the first two elements of the set and make the smallest of the two the current minimum and the largest the current maximum.
>
> Take the next two elements, compare them, and then compare the minimum with the current minimum and the maximum of the two with the current maximum. The total number of comparisons performed is 3 instead of 4 (2 per key).

Problem View the same problem as a divide-and-conquer problem.

Question How many comparisons do you need to find simultaneously the min and max of n keys?

Answer. At most $3n/2-2$.

Question. Suppose we have found the minimum. How many additional comparisons are required to find the second smallest key of the original n keys?

Answer. $\lceil \lg n \rceil - 1$.

or **Question.** Find the second smallest of n keys is $n + \lceil \lg n \rceil - 2$ comparisons.

Expected linear time algorithm

The idea of the algorithm is to use recursive partitioning (divide and conquer) just as quick sort did. The difference from quick sort is that in the recursive step only one of the two partitions needs to be further searched. This difference leads to an $\Theta(n)$ expected time algorithm rather than $\Theta(n \lg n)$.

The algorithm uses **Randomized-Partition** which is similar to **Partition** except that the splitter is chosen at random (instead of being the first element of the subsequence passed as an argument). Then **Randomized-Select** works as follows.

```
 Randomized-Select (A,l,r,i)
1. if (l=r) the return A[l]
2. m=Randomized-Partition(A,l,r);
3. k=m-l+1;// Line 3 computes the size of the left partition
4. if (i== k)  // Is statistic  A[m]?
5.  return(A[m]); //yes return it
5. elseif (i<k) //no, is it on left side?
6.   return Randomized-Select(A,l,m-1,i)  // yes it is
7. else
8.   return Randomized-Select(A,m+1,r,i-k) //no it is not
```

The worst case running time is $\Theta(n^2)$. For the average case the recurrence to solve is

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

whose solution is $T(n) = O(n)$.

This is so because if splitter is the minimum, then the left partition is of size 1 (prob $1/n$ this happening). If otherwise rank(splitter) ≥ 2, the left set is $1, 2, \ldots n-1$ with probability $1/n$ for each case as well. Therefore probability left set is 1 is $2/n$ (two cases this happening).

(c) Copyright A. Gerbessiotis. All rights reserved.

Worst case linear time

We describe an algorithm whose worst case running time is $O(n)$. We achieve this by splitting the input set more accurately than using Randomized-Partition. The algorithm below describes procedure **Select(A,i)**.

```
   Select (A,i) // Find i-th smallest in set A of n keys.
1. Group n keys into groups of  5 keys.
2. Sort the keys in each group with insertion sort.
3. Identify the median of each group and collect them in B.
4. Find the median x of the medians of the groups
   recursively using Select(B,floor((ceil(n/5)+1)/2)).
5. Partition A into
   Aleft  = { a: a belongs to A and a <  x }
   Aright = { a: a belongs to A and a >= x }
6. Let |Aleft|=k and |Aright|=n-k.
7. if (i<=k)
8.    Select(Aleft,i);
9. else
10.Select(Aright,i-k);
```

1. Split the n keys into groups of 5 keys. There are ceil($n/5$) many groups of which floor(n/5) have 5 keys and the remainder the remaining keys of the input set (less than 5).

2,3. Find the median of each group of 5 keys by sorting each group with say, insertion sort. This requires constant time $O(5^2) = O(1)$ per group and $O((n/5))$ overall, since insertion sort on 5 keys requires $O(1)$ time per group and thus $O(n)$ for all groups. The median of 5 keys is the third smallest (or largest) of the 5 keys.

4. Use Select recursively to find the median x of the ceil(n/5) medians of step 3.

5. Partition the input keys around x. Let k be the number of elements less than x and $n-k$ the number of elements greater than or equal to x forming respectively sets Aleft and Aright.

7-10. Use Select recursively to find the i-th smallest key in Aleft if $i \leq k$ or the $(i-k)$-th smallest key in Aright otherwise.

(c) Copyright A. Gerbessiotis.

The first step in the analysis of Select is the finding of the size of the set of elements which are larger than x. At least half of the medians in step 3 are greater than or equal to x. For each such median there are three elements (including the median element) out of 5 that are greater than x except for the last group (which may have fewer than 5 elements, i.e fewer than 3 greater than x) and the group containing x (which has 2 keys greater than x), plus the two elements greater than x. Therefore THE NUMBER OF KEYS GREATER THAN x is AT LEAST

$$3((1/2)\lceil n/5\rceil - 2) + 2 \geq 3n/10 - 4$$

Then, the NUMBER OF KEYS LESS THAN x is AT MOST

$$n - (3n/10 - 4) \leq 7n/10 + 4$$

Similarly, THE NUMBER OF KEYS LESS THAN x is AT LEAST $3n/10 - 4$ and NUMBER OF KEYS GREATER THAN x is AT MOST $7n/10 + 4$. Therefore in step 5 and in steps 8-10 Select is called on a set whose maximum size is at most $7n/10 + 4$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Graphical View of the keys

Assume we have sorted the medians of the groups of 5 left to right (smallest-to-largest) and also the keys within a group bottom-to-top (smallest to largest). Then the n keys look like.

```
                      -----------------------   Call the median of the medians x(=100)
       100   300   1010 |140  122  144  162
        70   200   1000 |110  121  143  161
      -----------------------                 >=x   The top   right area is the set of keys >=x
        30    60     80 \100| 120  140  160          The bottom left area is the set of keys <=x
<=x                      --/------------------      #keys (<=x) >= 3n/10 [bottom-left]
        29    10     70   /5   10    4    5          #keys (>=x) >= 3n/10 [top-right]
        19     5     20  / 1    2    3    4          HOWEVER WE WANT TO KNOW AN UPPER BOUND
     -------------------/                           ON THE NUMBER OF KEYS not the 3n/10 lower
                                                     bound.
```

If we use [bottom-left] we establish the upper-bound for the [top-right] area
Use #keys (<=x) + # keys (>x) =n i.e. #keys (>x) = n - #keys(<=x) <= n - 3n/10 = 7n/10

If we use [top-right]   we establish the upper-bound for the [bottom-left] area
Use #keys (<=x) + # keys (>x) =n i.e. #keys (<x) = n - #keys(>=x) <= n - 3n/10 = 7n/10

(c) Copyright A. Gerbessiotis.

We then analyze the performance of Algorithm Select in detail.

Steps 1,2 and 3 require $O(n)$ steps. Step 4 takes $T(\lceil (n/5) \rceil)$ steps and step 5-10 requires $T(7n/10+4)$ steps. Therefore If $n > 100$,

$$T(n) = T(n/5+1) + T(7n/10+4) + n \quad , \quad n \geq 100$$

else if $n < 100$, $T(n) = \Theta(1)$ i.e. it takes constant time to find the median or other statistic of a constant number (up to 100) number of keys. We solve the recurrence using the subistitution method. Let us guess $T(n) \leq cn$ for $n \geq 100$. Then

$$\begin{aligned} T(n) &= T(n/5+1) + T(7n/10+4) + n \leq c(n/5+1) + c(7n/10+4) + n \\ &= 9cn/10 + 5c + n \end{aligned}$$

We observe that $5c + n \leq cn/10$ for say $c = 20$ as long as $n \geq 100$. Large values of c are OK as well. Therefore, for $n \geq 100$

$$T(n) \leq 9cn/10 + 5c + n \leq 20n$$

which implies that $T(n) = O(n)$, as needed.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.