QuickSort, linear-time sorting

and the complexity of Sorting

Chapter 4 of GT

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 2016; distribution outside this group of students is NOT allowed.
Quicksort
Introduction

We examine another sorting algorithm, namely quicksort. It exhibits an interesting behavior among all sorting algorithms. Its worst-case running time performance is $\Theta(n^2)$. Despite this performance, it is one of the most practical choices of a sorting algorithm as its average case performance is $\Theta(n \lg n)$ with the constant factors hidden smaller than both mergesort and heap sort.

Quicksort is a very simple algorithm to describe. Its performance however it is more cumbersome to describe and analyze.

Quicksort is a Divide-and-Conquer algorithm as is mergesort. The three steps of a divide-and-conquer approach to sort an array of size $n$ are recursively shown for a subarray $A[l..r]$. In order to sort $A$ we issue a call Quicksort($A[0..n-1]$).

Quicksort($A[l..r]$)

**Divide**
Partition the elements of $A[l..r]$ into two nonempty subarrays/subsequences $A[l..m]$ and $A[m+1..r]$ such that all elements of the first subarray are smaller than or equal to $A[m]$ and all elements of the second subarray are at least $A[m]$. Index $m$ is not the middle element as in mergesort but is computed as part of the partitioning algorithm. Note that during this step, elements of $A$ move around but within the bounds of the subarray $A[l..r]$.

**Conquer**
Each of the two subarrays is sorted recursively by calling quicksort.

**Combine**
Everything is sorted as a result of the conquer phase (if one sorts $A[l..m]$ and then sorts $A[m+1..r]$, because of the partitioning in step Divide, $A[0..n-1]$ becomes sorted). Do nothing more and return.

Below we describe in detail the phases of quicksort, as originally proposed by its creator C.A.R. Hoare.

Because of its simplicity the major task in quicksort is to split $A[l..r]$ as evenly as possible so that $A[l..m]$ and $A[m+1..r]$ have approximately the same number of elements i.e. $m - l + 1 \approx r - m$. This means that the splitter element $A[m]$ (as all elements of $A[l..m]$ are at most $A[m]$ and all elements of $A[l..m]$ are at least $A[m]$) must be well chosen (i.e. be close to the median of the input sequence).
The `QuickSort` function is given below. (Note: the textbook picks the splitter to be the right-most key; we show here the same thing with the leftmost key).

```
QuickSort(A,0,n-1); // Sort array A[0..n-1]
```

The implementation of `QuickSort(A,1,r)` follows.

```
QuickSort(A,1,r);
1. if (1<r)
2. then m=PARTITION(A,1,r); // A[1,...,m-1,m,m+1,...r]
3. QuickSort(A,1,m); // sort left set.
4. QuickSort(A,m+1,r); // sort right set.
```

The implementation of `Partition(A,1,r)` follows.

```
Partition(A,1,r); // Sort A[1 ... r]
// Reorder so that A[1...j j+1...r]
// <= >= splitter
1. splitter = A[1]; // Leftmost key to become the splitter: store it in splitter
2. i= l-1; j=r+1; // i points to the left, j to the right of area to be sorted
3. while (1) {
4. do j=j-1; // Skip keys on the right if they are > splitter
5. while (A[j] > splitter); // stop if you find something <= splitter on the right
6. do i=i+1; // Skip keys on the left if they are < splitter
7. while (A[i] < splitter); // stop if you find something >= splitter on the left
8. if (i<j) swap(A[i],A[j]); // Swap a left key >= splitter with a right <= splitter
9. else return(j); // If i and j cross (i>=j) then return j
10 } 
```
Quicksort
Explanation

A description of the algorithm follows. The splitter or pivot key is the leftmost one of the sequence to be sorted \((A[l..r])\). Indices \(i\) and \(j\) scan the “left” and “right” areas of the area to be partitioned \((A[l..r])\).

We scan through the array starting from the rightmost element until we find a key \textbf{less than or equal to the splitter}. Its position is indexed by \(j\).

We scan through the array starting from the leftmost element until we find a key \textbf{greater than or equal to the splitter}. Its position is indexed by \(i\).

If \(i < j\) (i.e. \(i\) points to an element on the “left area” and \(j\) points to an element of the “right area”), then we swap \(A[i]\) and \(A[j]\). Otherwise, \(i\) and \(j\) crossed each other (i.e. \(i \geq j\) and \(i\) points to the “right area” and \(j\) to the “left area”). In that case \(j\) is the splitting position. All keys \(A[l..j]\) are less than or equal to the splitter and all keys in \(A[j+1..r]\) are greater than or equal to the splitter. The splitter can be on one or the other side (what does this say about the stability of quick-sort?)

One interesting by-product of this procedure is the case where all the keys are the same. What happens then to the input keys. How are they split? Explain.
**Quicksort**

A variant of the original algorithm

In the version of quicksort below, we ignore the splitter until the very end; for this reason \( i \) starts from \( l \) rather than \( l-1 \). This requires a change at the top-level call.

```plaintext
QuickSort(A,l,r);
1. if (l<r)
2. then m=Alternative_View_Partition(A,l,r); // A[l,...,m-1,m,m+1,...r]
3. QuickSort(A,l,m-1); // sort left set.
4. QuickSort(A,m+1,r); // sort right set.
```

The implementation of Alternative_View_Partition\((A,1,r)\) follows. If you implement both algorithms and benchmark them, the latter is going to be faster in practice.

```plaintext
Alternative_View_Partition(A,l,r); //
// If you have this Partition how can you change
// the two recursive calls to QuickSort?
1. splitter = A[l];
2. i= l ; j=r+1;
3. while (1) {
4.   do j=j-1;
5.   while (A[j] > splitter);
6.   do i=i+1;
7.   while (A[i] < splitter);
8.   if (i<j) swap(A[i],A[j]);
9. else {
11.      return(j);
12.     }
13 } 
```
Quicksort
Textbook Version

GT-QuickSort(A,1,r);
1. if (1<r)
2. then m=GT-PARTITION(A,1,r); \( A[1,...,m-1,m,m+1,...r] \)
3. QuickSort(A,1,m-1); \( \text{ sort left set.} \)
4. QuickSort(A,m+1,r); \( \text{ sort right set.} \)

GT_Partition(A,1,r); // Splitter is the rightmost key (similar to Alternative_View)
1. splitter = A[r];
2. i=1; j=r-1;
3. while (i<=j) {
4. while (i<=j) && (A[i] <= splitter) i++;
5. while (j>=i) && (A[j] >= splitter) j--;
6. if (i<j)
7. swap(A[i],A[j])
8. }
9. swap(A[i],A[r]);
10. return(i);

The only change from page 248, is that we split the original code by separating the top-level call from Partition.

One interesting observation is the behavior of this code when all the keys are equal. Performance-wise this code is between the previous two.

**Assessing the performance of quicksort** It is noted that the performance of function Partition is proportional to the number of elements to be partitioned ie \( (n = r - l + 1) \). Given that merge-sort is an \( O(n \log n) \) algorithm we expect quicksort to have the same performance as long as Partition splits the array into two partitions of equal size. An average case argument lends credibility to such an argument. Half of the elements in the average case will be smaller than the splitter and half larger.
Quicksort Performance
Worst-Best-Average case

50% - 50% split In each recursive step $n/2$ and $n/2$ are the sizes of the two subsequences (this is a very optimistic best-case).

1 vs $n - 1$ split We shall show that this as well as the symmetric one leads to worst-case performance.

$cn$ vs $n - cn$ split Assuming that $c < 1$ we shall show that this is still as good as the best-case.

average case To be examined.

Partitioning a sequence of size $n$ take $dn$ time for some constant $d$, and $T(1) = \Theta(1)$, where $T(n)$ the worst-case running time of quicksort.

\[
T(n) = T(n-1) + \Theta(n) \\
= T(n-k) + \Theta(n-k+1) + \ldots + \Theta(n) \\
= T(1) + \Theta(2) + \ldots + \Theta(n) \\
= T(1) + \Theta(2 + \ldots + n) = T(1) + \Theta(n^2) \\
= \Theta(n^2)
\]

Therefore the best-case of insertion sort is worst-case for quick sort.

Under the best-case the recurrence for running time is shown below.

\[
T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)
\]

The $cn$ split provides the recurrence relation shown below

\[
T(n) = T(cn) + T((1-c)n) + \Theta(n) = \Theta(n \lg n)
\]

Therefore, as long as $c$ is a constant, a constant fraction split leads to an algorithm that is $\Theta(n \lg n)$. The closer the constant $c$ is to $1/2$ the smaller the constant in the $\Theta$ notation is.
To analyze the average case of quicksort we assume that all permutations of the input are equally likely to occur. In the average case, we would have a mix of **bad** and **good** splits. For example if a bad split \(1\) and \(n - 1\) is followed by a good split, the second split leads to an \((n - 1)/2\) and \((n - 1)/2\) split of the \(n - 1\) keys. This gives a split of \((n - 1)/2 + 1\) and \((n - 1)/2\) after two iterations.
**QuickSort**

**Worst-case running time of GT-QuickSort**

In general the performance of quicksort is given by the recurrence, where $c$ is some predefined positive constant $c$ determined by GT-Partition.

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

Then, the worst case performance of quicksort is

$$T(n) = \sum_{i=0}^{n-1} (T(i) + T(n - i - 1)) + n$$

We will show that this implies that $T(n) = \Theta(n^2)$ by using the substitution method. This is parts (a), (b) below.

(a) Show that $T(n) = O(n^2)$, i.e. $T(n) \leq cn^2$ for some $c > 0$ and for all $n \geq n_0$. We use the fact that $i^2 + (n - 1 - i)^2$ is maximized for $i = 0$ or $i = n - 1$ (i.e. for one of the boundary values). Then

$$T(n) = \sum_{i=0}^{n-1} (T(i) + T(n - i - 1)) + n$$

$$\leq \sum_{i=0}^{n-1} (c_i^2 + c(n - i - 1)^2) + n$$

$$= c \cdot \sum_{i=0}^{n-1} (i^2 + (n - i - 1)^2) + n$$

$$\leq c \cdot (n - 1)^2 + n$$

$$= c(n^2 - 2n + 1) + n$$

$$\leq cn^2,$$

as long as $c \geq 1$ and $n \geq 1$, since then $c + n \leq cn + cn = 2cn$. 

(c) Copyright A. Gerbessiotis. CS610: Spring 2016. All rights reserved.
QuickSort
Worst-case performance continued

(b) Exercise. Show similarly that $T(n) = \Omega(n^2)$, i.e. $T(n) \geq fn^2$ for some $f > 0$ and for all $n \geq n_1$ (for some positive $n_1$).

Exercise. What happens with the best-case performance of quicksort? How can we show that $T(n) = \Omega(n \lg n)$ or a tight $\Theta$ bound? We can work similarly to the worst case. The only change will be finding not the maximum of $f(i) = i^2 + (n - i)^2$ but the MINIMUM of function $f(i) = i \lg i + (n - i) \lg (n - i)$. The latter has a minimum at $i = n/2$, which shows that the best case performance is attained for a 50–50 split.
Randomized QuickSort

A randomized version of quicksort, before the execution of the algorithm, randomizes the position of the keys so that the average case be observed. Note that in this version of quicksort the worst case or base case are rarely observed. Their appearance depends on the sequence being so unlucky that the random numbers generate a sorted sequence out of the input sequence.

A randomized algorithm is one whose behavior is also determined (besides the input) by the values of a random number generator. This way bad inputs don’t necessarily lead to worst-case running time. The disadvantage is that in some case good inputs may lead to worst-case running time. However the chances of this occurring are negligible. And even if they occur (we can count number of operations to detect such a case) we always have a quick fix. Stop the current execution, reobtain the original sequence, rerandomize it and hope that the second time around we would be luckier.

Another way to proceed is to change the partition method: The splitter becomes a key in a random position instead of the key in position 1.

```
Randomized-Partition (A, l, r)  Rand-QuickSort(A, l, r)
1. i <- Random(l, r)  1. If (l < r)
2. exchange A[i] and A[r]  2. m = Randomized-Partition(A, l, r)
3. return GT-PARTITION(A, l, r)  3. Rand-Quicksort(A, l, m-1)
4. 4. Rand-Quicksort(A, m+1, r)
```
QuickSort
Randomized Case analysis

From the quicksort recurrence, for \( i = 0 \) to \( i = n - 1 \), we have \( T(n) = T(i) + T(n - 1 - i) + n \). The splitter is the \( i + 1 \)-st smallest element (i.e. the one whose index is \( i \)) of the sequence induced by Partition. Since \( i + 1 \) takes values from 1 (inclusive) through \( n \) (inclusive), we need to figure out the probability that a given split occurs.

Say the splitter is the smallest key \( (i = 0) \). The probability this happening is obviously \( 1/n \). This yields a 0 and \( n - 1 \) split. Say the splitter is the largest of the \( n \) keys \( (i = n - 1) \). The probability this happening is obviously \( 1/n \). This yields an \( n - 1 \) and 0 split. Say the splitter is the 2nd, 3rd, all the way to \( n - 1 \)st largest of the \( n \) keys i.e. anything but the smallest or the largest. The probability that this happening is also \( 1/n \) since we have \( n \) keys altogether. Consider what happens in that case: the splitter behaves similarly to case two, when it was the largest of the \( n \) keys. The left set is of size 1, 2, \( \ldots, n - 2 \) with the same probability \( 1/n \).

Therefore the left set is of size 0, 1, 2, \( \ldots, n - 1 \) with probability \( 1/n \).

Observation 1. We first note that \( T(0) = T(1) = 0 \).

Observation 2. We know that \( H_n = 1 + 1/2 + \ldots + 1/n \approx \ln n \), and thus \( 1/3 + \ldots + 1/(n+1) = H_{n+1} - 3/2 \).

Then the recurrence of the running time is given below.

\[
T(n) = \frac{1}{n} \left( \sum_{i=0}^{n-1} T(i) + T(n - i - 1) \right) + n
\]
QuickSort
Randomized Case analysis

So far we have shown that,

\[ T(n) = \frac{1}{n} \left( \sum_{i=0}^{n-1} T(i) + T(n - i - 1) \right) + n \]

We multiply by \( n \) both sides and then substitute in the resulting equality \( n - 1 \) for \( n \). We write both derivations below

\[ nT(n) = \sum_{i=0}^{n-1} (T(i) + T(n - 1 - i)) + n^2. \]

\[ (n - 1)T(n - 1) = \sum_{i=0}^{n-2} (T(i) + T(n - 2 - i)) + (n - 1)^2. \]

We subtract the second from the first equality. We get after we divide both sides by \( n(n + 1) \).

\[ nT(n) - (n - 1)T(n - 1) = 2T(n - 1) + n^2 - (n - 1)^2 \]

\[ nT(n) - (n - 1)T(n - 1) = 2T(n - 1) + (2n - 1) \]

\[ T(n) = \frac{T(n - 1) + (2n - 1)}{n + 1} \]

Now let us substitute \( t(n) = \frac{T(n)}{n + 1} \). We get

\[ \frac{T(n)}{n + 1} = \frac{T(n - 1)}{n} + \frac{(2n - 1)}{n(n + 1)} \]

\[ t(n) = t(n - 1) + \frac{(2n - 1)}{n(n + 1)} \]

By Observation 1 we have \( t(0) = t(1) = 0 \).
**QuickSort**

**Randomized Case analysis**

We then upper bound the last term. An alternative method is expand the last term as in \( \frac{2n-1}{n(n+1)} = -\frac{1}{n} + \frac{3}{n+1} \).

\[
t(n) = t(n-1) + \frac{2n-1}{n(n+1)}
\]

Expanding the recurrence (iteration method) we get

\[
t(n) = t(n-1) + \frac{-1}{n} + \frac{3}{n+1}
\]

\[
t(n) = t(1) + \left(\frac{-1}{2} + \ldots + \frac{1}{n}\right) + 3\left(\frac{1}{2} + \ldots + 1/(n+1)\right)
\]

\[
t(n) = -(1/2 + \ldots + 1/n) - 3 + 3(1 + 1/2 + \ldots + 1/n + 1/(n+1))
\]

\[
t(n) = 2(1 + 1/2 + \ldots + 1/n) + 3/(n+1) - 3
\]

\[
t(n) = 2H_n + \frac{3}{n+1} - 3
\]

\[
T(n)/(n+1) = 2H_n + 3/(n+1) - 3
\]

\[
T(n) = 2(n+1)H_n + 3 - 3(n+1)
\]

\[
T(n) \leq 2n \ln n
\]

The last one was derived because from \( H_n = \ln n + \gamma \) we have \( 2(n+1)H_n + 3 - 3(n+1) - 2n \ln n = 2n \ln n + 2n\gamma + 2H_n - 3 - 3(n+1) \leq 0 \). From the last two derivation we conclude \( T(n) \approx 2n \ln n \).
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/Bucket-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 $O(d(n + k))$.

**Probabilistic-Sort/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** $O(n)$.

**Note.** Count-sort is sometimes called Bin-sort and rarely Bucket-sort. The textbook however follows this convention. In order to avoid any confusion we will follow the textbook and WILL NOT spend any time discussing the Probabilistic-Sort algorithm sometimes ALSO called Bucket-Sort.
Complexity of Sorting

Lower bounds

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.
Lower bounds of Sorting
A three-key decision tree for InsSort

Input is \(<a_1 \ a_2 \ a_3>\) We run Insertion Sort

```
     ________
    |a1:a2|
   < / \ >
  _____ / _
 |a2:a3|     |a1:a3|
_____/ \____
 |a1:a3|     |a2:a3|
 </ \>     </ \>
  _\_____  _\_____
 <a1 a2 a3> |a1:a3| <a2 a1 a3> |a2:a3|
     / \     / \     / \     / \     / \     / \
 <a1 a3 a2> <a3 a1 a2> <a2 a3 a1> <a3 a2 a1>
```

All 3!=6 permutations of the input sequence appear as leaves.
If \(<a_1 \ a_2 \ a_3>\) is the reached leaf it means that in the input
\(<a_1 \ a_2 \ a_3>\) we have that \(a_1 < a_2 < a_3\)
If however \(<a_3 \ a_2 \ a_1>\) is the reached leaf it means that in the
input \(a_3 < a_2 < a_1\)
Lower bounds of Sorting
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 \( n! \geq \left( \frac{n}{e} \right)^n n \). This implies that \( h \geq n \lg n - n \lg e = \Omega(n \lg n) \).

**Corollary.** Heapsort and mergesort are asymptotically optimal.

**Exercise.** Can we sort 5 arbitrary keys with at most 7 comparisons in the worst-case? Explain.

**Note.** For Binary Search a sorted array of \( n \) keys, if the key is in the array one of the \( n \) indices is to be returned to show the position of the key; if the key is not in the array a NOT-FOUND value is to be returned. There are a total of \( n + 1 \) possible outcomes. One can use a decision tree argument to show that searching with comparisons takes at least \( \lfloor \lg n \rfloor + 1 = \lceil \lg (n + 1) \rceil \) comparisons.
Linear time sorting

BucketSort/Countsort

Let us assume that \( n \) keys to be sorted take values from a set \( 0, \ldots, k - 1 \). We are going to present an algorithm whose running time is \( O(n + k) \). Then IF \( k = O(n) \) this is \( O(n) \), ie a linear time algorithm for sorting exists. Note that the linearity or not of the performance of the algorithm depends on \( k \) The optimal \( k \) is when the number of values of the keys is about (i.e. asymptotically equal to) the number of keys.

The main idea for this “often linear-time” sorting algorithm is that since the size of the set of values of a key is ”small” we can count the number of times each key value appears in the sorted sequence and we are going to “place” all keys having the same value into the same “bucket”. Then we only need to concatenate the buckets to sort the \( n \) keys by first concatenating the buckets (i.e. attaching the lists) of the keys having values 0 and 1, then concatenating those keys with value 2 and so on. Concatenation of linked lists can take constant time if we maintain tail pointers. However even this bucket creation process is not required.

```
Bucket-Sort(I,O,n,k) // I[0..n-1] is input  O[0..n-1] is output
1. for(i=0;i<k;i++)
2.  InitializeListforBucket(Bucket[i]); // Create k buckets Bucket[i] initially NULL
   // Let us assume Bucket[i].length is number of elts in bucket i.
3. for(j=0;j<n;j++)
4.  AttachKeyToBucket(A[j],Bucket[A[j]]);
5. Initialize(O,n) // Make sure output O is empty
6.  count=0;
7.  for(i=0;i<k;i++)
8.  Set(O,count,Bucket[i].length); //All keys O[count,..., count+Bucket[i].length-1] are i
9.  count = count +Bucket[i].length;
10. return(O);
```
Linear time sorting
BucketSort/Countsort

The algorithm below even eliminates this bucket creation process by only counting the keys having a specific value. Therefore it is more appropriate to call it Count-Sort rather than Bucket-Sort.

Count-Sort(I,0,n,k)  // I[0..n-1] is the input array 0[0..n-1] is the output array
1. for(i=0;i<k;i++)
2.   C[i]=0;  // Initialize the Counter Array of size k
3. for(j=0;j< n;j++)
4.   C[I[j]] ++;  // If key I[j] is m increment C[m] by one
   // Result: C[t] : number of keys with value t
5. for(i=1;i<k;i++)  // Make C[i] to hold number of keys with values at most i.
6.   C[i] = C[i]+C[i-1];  // C[i]-1 is the last position in the output to be occupied by an i.
7. for(j=n-1;j>=0;j--)
8.   C[I[j]]--;  // Note from before C[I[j]]-1 is the index!
9.   O[C[I[j]]]=I[j];  // Store there the last I[j] and go on!
Linear time sorting
Analysis of Countsort

**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 $\Omega(n \lg n)$ lower bound fails to hold as count sort inspects values of keys and DOES NOT USE comparisons for sorting. Therefore when we talk about lower-bounds we need to be very explicit about the model of computation involved. The $\Omega(n \lg n)$ lower bound applies to a model of computation in which sorting involves comparison and exchange of keys. If counting of keys is used (as it was the case for Count-Sort or Bucket-Sort) then that lower bound fails to apply!

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)**

<table>
<thead>
<tr>
<th>Single Phase</th>
<th>Single Phase</th>
<th>Single Phase</th>
<th>Single Phase</th>
<th>Single Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 1 2</td>
<td>1 2 4</td>
<td>8 1 2</td>
<td>8 1 2</td>
<td></td>
</tr>
<tr>
<td>2 9 5</td>
<td>2 9 5</td>
<td>1 2 4</td>
<td>1 2 4</td>
<td></td>
</tr>
<tr>
<td>1 2 4</td>
<td>8 1 2</td>
<td>2 9 5</td>
<td>2 9 5</td>
<td></td>
</tr>
</tbody>
</table>

**Right To Left Operation (IT SORTS)**

<table>
<thead>
<tr>
<th>Single Phase</th>
<th>Single Phase</th>
<th>Single Phase</th>
<th>Single Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 1 2</td>
<td>8 1 2</td>
<td>8 1 2</td>
<td>1 2 4</td>
</tr>
<tr>
<td>2 9 5</td>
<td>1 2 4</td>
<td>1 2 4</td>
<td>2 9 5</td>
</tr>
<tr>
<td>1 2 4</td>
<td>2 9 5</td>
<td>2 9 5</td>
<td>8 1 2</td>
</tr>
</tbody>
</table>
Radix Sort
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 perform 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.
Radix Sort
Applications

A. Bits and Values

Fact 0. The integer numbers 0, . . . , m − 1 can be represented by ⌈lg m⌉ bits.

Fact 1. With x bits we can represent 2^x distinct non-negative integer values: 0, 1, . . . , 2^x − 1.

Fact 2. Therefore, with lg y bits we can represent 2^(lg y) = y values 0, . . . , y − 1.

Fact 3. What is 110?

• Base 2 (radix-2, binary) 110 is 110 = 1 · 2^2 + 1 · 2^1 + 0 · 2^0 = 6.

• Base 10 (radix-10, decimal) 110 is 110 = 1 · 10^2 + 1 · 10^1 + 0 · 10^0 = 110.

• Base 16 (radix-16, hexadecimal) 110 is 110 = 1 · 16^2 + 1 · 16^1 + 0 · 16^0 = 272.

Fact 4. Countsort sorts n keys in the range 0, . . . , k − 1 in time O(n + k).

Fact 5. Count-Sort sorts n keys in the range 0, . . . , k^d − 1 in time O(n + k^d).

Fact 6. Radix-Sort sorts n keys in the range 0, . . . , 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 · (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 · (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 · (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 · (n + 2^1)).
Radix Sort
Applications (continued)

**Fact 8.** We have $n$ keys in the range $0, \ldots, n^2 - 1$. How do 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 do 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 $O\left(\frac{n}{\lg n}(n + k)\right) = O\left(\frac{n}{\lg n}(n + n)\right) = O(n^2/\lg n)$. 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 $\text{Prob}(Y_{ij} = 1) = 1/n$ and thus $\text{Prob}(Y_{ij} = 0) = 1 - 1/n$.
5. $E[Y_{ij}] = 1 \cdot \text{Prob}(Y_{ij} = 1) + 0 \cdot \text{Prob}(Y_{ij} = 0) = 1(1/n) + 0(1 - 1/n) = 1/n$ because of 2.
6. $E[Y_{ij}^2] = 1^2 \cdot \text{Prob}(Y_{ij} = 1) + 0^2 \cdot \text{Prob}(Y_{ij} = 0) = 1(1/n) + 0(1 - 1/n) = 1/n$ because of 2.
7. $\text{var}(Y_{ij}) = E[Y_{ij}^2] - E^2[Y_{ij}] = 1/n - 1/n^2 = (1/n)(1 - 1/n)$. 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. $\text{var}[n_i] = \text{var}[\sum_j Y_{ij}] = \sum_j \text{var}[Y_{ij}] = n(1/n)(1 - 1/n) = 1 - 1/n$ 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 $k, k \neq j$.
Probabilistic Sort
The Algorithm

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

Sometimes we call this Probabilistic Sort algorithm Bucket-Sort.

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 \(Bucket[0..n-1]\).

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

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 \(Bucket[i]\)). Each key as it is drawn uniformly at random has probability \(p_i = 1/n\) to fall into \(Bucket[i]\). 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

\[
T(n) = \sum_{i=0}^{n-1} O(n_i^2) = O\left(\sum_{i=0}^{n-1} n_i^2\right)
\]

Then, by taking expectations on both sides we get

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