

QUICKSORT, LINEAR-TIME SORTING

and the complexity of Sorting

Chapters 8.2-8.3, 9.1, 10.3 of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Quicksort

Introduction

We introduce and examine another sorting algorithm, namely quicksort. Quicksort is a very simple algorithm to describe. Its performance however it is more cumbersome to describe and analyze. 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 heapsort. Furthermore, quicksort is a divide-and-conquer algorithm as was mergesort.

Method 1 (Quicksort). Quicksort is a sorting algorithm that utilizes divide and conquer. Even if divide and conquer naturally provides a recursive solution, one can eliminate recursion by providing an iterative solution for quicksort through the use of two stacks (or one stack holding two index boundaries at a time). The divide, conquer and combine phase of quicksort are highlighted below.

- **Divide phase:** partition n keys into two sequences L and R around a splitter (also known as pivot) key. Into set L put the keys that are no more than the splitter. The remaining keys go into R .
- **Conquer phase:** Likewise (recursively) sort the two sequences generated.
- **Combine phase:** Concatenate the two sorted sequences.

Quicksort is neither stable nor does it sort in-place.

Below we describe in detail the phases of quicksort, as originally proposed by its creator C.A.R. Hoare. It resembles the top level structure of MergeSort.

Method 2 (QuickSortCARH). The code is shown below. The pivot key is the left-most key of the input sequence.

Because of its simplicity the major task in quicksort is to realize in particular function CARH and in it to split subsequence $A[l..r]$ as evenly as possible so that $A[l..m]$ and $A[m+1..r]$ have approximately the same number of keys i.e. $m-l+1 \approx r-m$. The splitter keys must be chosen carefully for that. In practice it is not: we either pick the leftmost, or the rightmost or the middle input keys as the splitter. After the splitter is picked and rearrange the input keys, all keys of $A[l..m]$ are at most the splitter, and all keys of $A[m+1..r]$ are at least the splitter.

```

QuickSortCARH(A[0..n-1],n)
1.  CARH(A,0,n-1);

    CARH(A,l,r)    // A[l..r]
1.  if ( l < r ) {
2.      m=PARTITION(A,l,r); // A[l,...,m-1,m,m+1,...,r]
3.      CARH(A,  l, m);    // sort L = A[l .. m]
4.      CARH(A, m+1, r);    // sort R = A[m+1 .. r]

    PARTITION(A,l,r);      // Reorder so that A[l...j j+1...r]
                           //          <=      >= splitter
1.  splitter = A[l];        // Leftmost key is the splitter
2.  i= l-1; j=r+1;          // i points to left of left, j to right of right
3.  while (1) {
4.      do j=j-1;            // Skip keys on right    > splitter
5.      while (A[j]>splitter); // stop if key is found <= splitter
6.      do i=i+1;            // Skip keys on left     < splitter
7.      while (A[i]<splitter); // stop if key is found >= splitter
8.      if (i<j) swap(A[i],A[j]); // Swap keys as long as i<j
9.      else return(j);       // If i and j cross (i>=j) then return j
10 }

```

Quicksort

CARH sort continued

Proof. 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 key until we find a key **less than or equal to the splitter**. Its position is indexed by j . We scan through the array starting from the leftmost key until we find a key **greater than or equal to the splitter**. Its position is indexed by i . If $i < j$ (i.e. i points to an key on the “left area” and j points to an key 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. As a side note the splitter because of line 7 could be moved to the right side. 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. ■

DRAFT

Quicksort

A variant of the original algorithm

We highlight below a slight variant of the quicksort algorithm described in the previous page.

Method 3 (QuickSortAlt). The code is shown below. The pivot key is the left-most key of the input sequence.

Proof. In the version of quicksort below, we ignore the splitter until the very end; for this reason i starts from l rather than $l - 1$. Moreover when partition returns, j in AltPartition or m in Alt indicate also the position of the splitter. Thus $A[m]$ is the splitter itself. We can ignore it in the recursive calls of lines 3 and 4 of Alt.

```
QuickSortAlt (A[0..n-1],n)
1.  Alt(A,0,n-1);

    Alt(A,l,r)  // A[l..r]
1.  if ( l < r ) {
2.      m=AltPartition(A,l,r); // A[l,..,m-1,m,m+1,..,r]
3.      Alt (A,  l, m-1);      // sort L = A[l .. m-1]
4.      Alt (A, m+1, r );      // sort R = A[m+1 .. r ]

    AltPartition(A,l,r);      // Reorder so that A[l...j j+1...r]
                                //          <= splitter
1.  splitter = A[l];           // Leftmost key is the splitter
2.  i= l  ; j=r+1;             // i points to left      , j to right of right
3.  while (1) {
4.      do j=j-1;               // Skip keys on right    > splitter
5.      while (A[j]>splitter); // stop if key is found <= splitter
6.      do i=i+1;               // Skip keys on left     < splitter
7.      while (A[i]<splitter); // stop if key is found >= splitter
8.      if (i<j) swap(A[i],A[j]); // Swap keys as long as i<j
9.      else{
10.         A[l]=A[j];
11.         A[j]=splitter;
12.         return(j);
13.     }
14. }
```

■

Quicksort

Textbook GTQuicksort

We compare the divide and conquer approach for the sorting algorithm MergeSort described in Subject 1 and the sorting algorithm GTQuicksort to be described on the next page.

	MergeSort	GTQuicksort
Divide phase:	Easy: Middle Point Split	Difficult: GTPartition
Conquer phase:	recursive or iterative	recursive or iterative
Combine :	Difficult: Merging	Easy: Nothing:Concatenate

The easy phase of MergeSort is the complicated one for QuickSort. The easy phase of QuickSort is the complicated one for MergeSort. The complicated phase of MergeSort is Merging (of two sorted sequences into one, a linear-time operation). The complicated phase of QuickSort is Partition (of the input keys participating in the current recursive call around a splitter key, also known as pivot key so that at the completion of Partition keys less than or equal to the splitter are on one side of the input/output array and the keys greater than or equal to the splitter are on the other side of the input/output array), is also a liner time operation.

DRAFT

Quicksort

Textbook GTQuickSort

		Time
GTQuickSort (A[0..n-1],n)		T(n)=
1. GT(A,0,n-1);		T(n)=
GT(A,l,r) // A[l..r]	l=0; r=n-1	
1. if (l < r) {	Theta(1)	
2. m= GTPartition(A,l,r); // A[l,...,m-1,m,m+1,...,r]	Theta(n)	
3. GT (A, l, m-1); // sort L = A[l .. m-1]	T(m)	
4. GT (A, m+1, r); // sort R = A[m+1 .. r]	T(n-m-1)	
GTPartition(A,l,r); // Reorder so that A[l...j j+1...r]		
	// <= >= splitter	
1. splitter = A[r];	// Rightmost indexed key is the splitter (A[m])	
2. i= l ; j=r-1;	// i points to left index l, j to left of r i.e. r-1	
3. while (i<=j) {		
4. while (i<=j) && (A[i] <= splitter) i++;	// find key on left > splitter	
5. while (j>=i) && (A[j] >= splitter) j--;	// find key on right < splitter	
6. if (i<j)	// if indeed found	
7. swap(A[i],A[j])	// then swap them	
8. }		
9. swap(A[i],A[r]);	// before return move splitter	
10. return(i);	// return A[i] is splitter then!	

The only change from page 248 of the textbook, is that we split the original code by separating the top-level call from Partition. GT: Goodrich Tamassia (aka textbook) version.

Method 4 (GTQuickSort). The pseudocode is on the previous page. The pivot key is the rightmost key of the input sequence.

Proof. One interesting observation is the behavior of this code when all the keys are equal. Performance-wise this code is between the previous two. It is noted that the performance of function **GTPartition** is proportional to the number of elements to be partitioned i.e. $(n = r - l + 1)$. Given that merge-sort is an $O(n \lg n)$ algorithm we expect quicksort to have the same performance as long as **Partition** splits the input array into two partitions of equal size. An **average** case claim lends credibility to such an argument. Half of the elements in the average case will be smaller than the splitter and half larger. However average case performance does not determine the running time; it is worst-case performance and in the worst-case the splitter can be the smallest or largest of the input keys. In that case the depth of the recursion can be $n - 1$ leading to quadratic performance. ■

DRAFT

Quicksort

GTPartition example

Example 1. Let the input be $A = \langle 5, 20, 15, 31, 20, 30, 20, 17, 20 \rangle$. We then run $\text{GTPartition}(A, 0, 8)$. We have then.

```

l=0                      r=8
 0  1  2  3  4  5  6  7  8
 5 20 15 31 20 30 20 17 20 =splitter
i=0                      j=8
  i=1
    i=2
      i=3x          j=7
5 20 15 31 20 30 20 17 20=splitter
5 20 15 17 20 30 20 31 20=splitter
    i=3
      i=4
        i=5x  j=7
          j=6
            j=5
              j=4x
5 20 15 17 20 30 20 31 20
          i=5
5 20 15 17 20 20 20 31 30
<-  <=20---> ^^ <->=20->
          ^^
          splitter @ i=5
          return(i): i=5

```

Question 1. Based on the previous example and the work of GTQuickSort , is GTQuickSort stable? (Hint: Look at the 20s.)

$$T(n) = T(m) + T(n - m - 1) + \Theta(n) \quad \text{running time} \quad (1)$$

$$C(n) = C(m) + C(n - m - 1) + n - 1 \quad \text{comparison count} \quad (2)$$

Best Case Performance :

$$T(n) = \min_{0 \leq m \leq n-1} \{T(m) + T(n - m - 1) + \Theta(n)\} \Rightarrow T(n) = \Theta(n \lg n).$$

Worst Case Performance :

$$T(n) = \max_{0 \leq m \leq n-1} \{T(m) + T(n - m - 1) + \Theta(n)\} \Rightarrow T(n) = \Theta(n^2)$$

Average Case Performance :

$$T(n) = \frac{1}{n} \left\{ \sum_{m=0}^{n-1} T(m) + T(n - m - 1) + \Theta(n) \right\} \Rightarrow T(n) = \Theta(n \lg n)$$

Therefore running time is: $T(n) = O(n^2)$.

Quicksort GTQuickSort performance

Proposition 1 (GTQuickSort). *The running time of GTQuickSort is $T(n) = O(n^2)$.*

Proof. The performance of $\text{GTQuickSort}(A, n)$ is determined by the performance of $\text{GT}(A, 0, n-1)$. Let $T(n)$ be the performance of the latter for n keys. For a recursive call such as $\text{GT}(A, l, r)$ the running time will depend on l and r and in particular the keys between l and r i.e. $T(r - l + 1)$.

For $\text{GT}(A, l, r)$ when it is invoked with $l=0$ and $r=n-1$, the running time of line 1 is constant $\Theta(1)$. No comparison is performed as $l < r$ is in fact resolved by a subtraction. The running time of line 2 is $\Theta(n)$ (for $l = 0, r = n - 1$), and the running time of line 3 is $T(m)$ and of Line 4 $T(n - m - 1)$. This is because the number of keys in $A[l..m - 1]$ and those of $A[m + 1..r]$ is m and $n - m - 1$ respectively for $l = 0$ and $r = n - 1$. (Be reminded that the splitter does not participate into further recursive calls.) Therefore the recurrence for the running time is summarized below, with the last term absorbing $\Theta(1)$ contributions.

$$T(n) = T(m) + T(n - m - 1) + \Theta(n) \quad (3)$$

Best case performance. For best case performance the right hand side's contribution is minimized in every iteration. Thus

$$\text{Best Case Performance : } T(n) = \min_{0 \leq m \leq n-1} \{T(m) + T(n - m - 1) + \Theta(n)\}$$

Worst case performance. For worst case performance the right hand side's contribution is maximized in every iteration. Thus

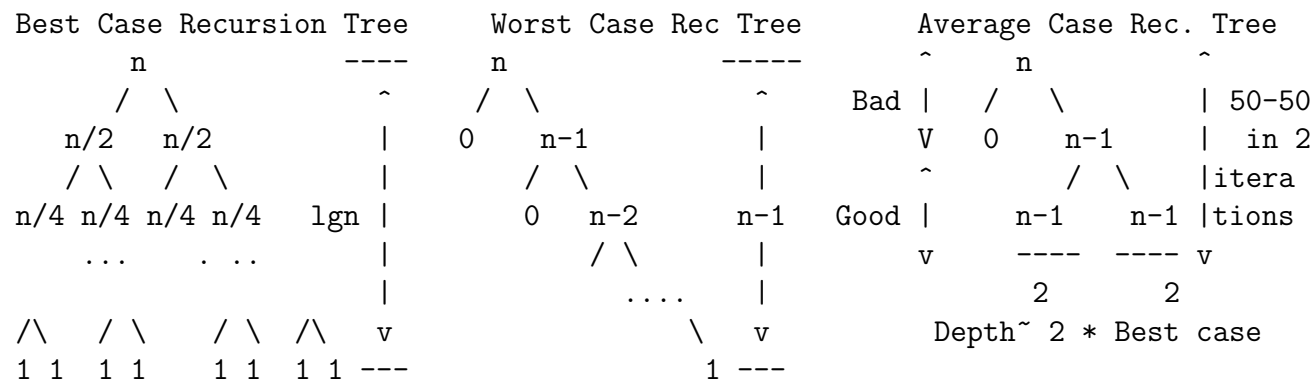
$$\text{Worst Case Performance : } T(n) = \max_{0 \leq m \leq n-1} \{T(m) + T(n - m - 1) + \Theta(n)\}$$

Average case performance. For average case performance we use that all splits are equally likely and average the possibilities.

$$\text{Average Case Performance : } T(n) = \frac{1}{n} \left\{ \sum_{m=0}^{n-1} T(m) + T(n - m - 1) + \Theta(n) \right\}$$

One way to visualize quicksort's extreme cases is to visualize the possibilities of the recursive calls for such cases. Next we show the recursion trees of the recursive calls for the best and the worst case and surmise a recursion tree for the average case. The worst-case scenario determines the running time and thus the proposition follows. (Details on the next two pages.) ■

Quicksort GTQuicksort three case performance visualization



Worst Case and Worst Case Performance. We can and will prove analytically that worst case performance is when the splitter is the smallest or the largest input key and this occurs in every iteration. Then the running time recurrence become $T(n) = T(n - 1) + \Theta(n)$ as one of the two T terms becomes $T(0)$ i.e. $\Theta(1)$. The solution is $T(n) = \Theta(n^2)$. The recursion is then of depth $n - 1$ and one side of the splitting contains 0 keys and all the remaining keys are on the other side. The recursion tree above depicts the case where the splitter is the minimum in every iteration. (A symmetric tree is for the maximum; a zig-zag tree is observed if we have alternations of min and max as a splitter.)

Average Case and Average Case Performance. 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. (A bad split is at worst a 0 and $n - 1$ split, a good split is at best a 50-50 split.) On the average a bad split is followed by a good split: the worst split can be at worst a 0 and $n - 1$, the subsequent good split is $(n - 1)/2$ and $(n - 1)/2$, i.e. we have a 50-50 split AFTER EVERY TWO iterations, on the average. This leads to an argument that the recursion tree has expected depth $2 \lg n$ and thus average case performance is twice as bad as best case performance but asymptotically equal and equal to $\Theta(n \lg n)$.

We prove the worst-case result through analytical means on the next page.

Quicksort

GTQuickSort worst-case performance analysis

Theorem 1. *The worst case running time of quicksort (GTQuickSort) can be shown to be $T(n) = \Theta(n^2)$. The worst-case is a sorted or a reverse-sorted sequence.*

Proof. The recurrence for the worst-case running time

$$T(n) = \max_{0 \leq m \leq n-1} \{T(m) + T(n - m - 1) + \Theta(n)\}$$

can be solved to yield two insights: (a) running time for the worst-case is $T(n) = \Theta(n^2)$, and (b) the worst case is when one side has 0 keys and the other side the remaining keys i.e. when the smallest or the largest of the keys participating in a recursive call is chosen to become the splitter. We can solve the recurrence and show $T(n) = \Theta(n^2)$ by using the substitution method. We show only the O part first i.e. $T(n) = O(n^2)$, that is $T(n) \leq cn^2$ for any $c \geq 1$ and for all $n \geq n_0 = 1$. We use the fact that $i^2 + (n - 1 - i)^2$ is maximized for $i = 0$ or $i = n - 1$ (i.e. at the boundary). This implies that in the worst case one of the two sides is of size 0 i.e. when the splitter is the maximum or the minimum. Then

$$\begin{aligned} T(n) &= \max_{0 \leq m \leq n-1} (T(m) + T(n - m - 1)) + n \\ &\leq \max_{0 \leq m \leq n-1} (cm^2 + c(n - m - 1)^2) + n \\ &= c \cdot \max_{0 \leq m \leq n-1} (m^2 + (n - m - 1)^2) + n \\ &\leq c \cdot (n - 1)^2 + n \\ &\leq cn^2 + (1 - 2c)n + cn \\ &= cn^2 + (1 - c)n \\ &\leq cn^2 \end{aligned}$$

The last inequality is so for any $c \geq 1$ and induction completes. The details for the $T(n) = \Omega(n^2)$ part are left as an exercise.

Quicksort

GTQuicksort worst-case performance analysis

Exercise 1. Show similarly to the $O(n^2)$ case, that for the worst-case $T(n) = \Omega(n^2)$, i.e. $T(n) \geq fn^2$ for some constant $f > 0$ and for all $n \geq n_1$ (for some positive n_1).

Then we prove similarly that

$$\begin{aligned} T(n) &= \max_{q=0}^{n-1} (T(q) + T(n - q - 1)) + n \\ &\geq \max_{q=0}^{n-1} (fq^2 + f(n - q - 1)^2) + n \\ &\geq f((n - 1)^2) + n \\ &= fn^2 + (1 - 2f)n + f \\ &\geq fn^2 + (1 - 2f)n \\ &\geq fn^2 \end{aligned}$$

The last inequality is derived provided that $1 \geq f/2$, i.e. $f \leq 2$, and $f > 0$. ■

Quicksort

GTQuickSort best-case performance analysis

In order to derive the best case performance of GTQuickSort we minimize the contribution of the right hand side in every iteration. Thus

$$T_{best}(n) = \min_{0 \leq m \leq n-1} \{T_{best}(m) + T_{best}(n - m - 1) + \Theta(n)\}$$

We can prove analytically that the best case is a 50-50 split. Then $T_{best}(n) = T_{best}(n/2) + T_{best}(n/2) + \Theta(n) = 2T_{best}(n/2) + \Theta(n)$, and the master method yields $T_{best}(n) = \Theta(n \lg n)$. (MergeSort properties are also a reminder.)

Theorem 2. *The best case running time of GTQuickSort can be shown to be $T(n) = \Omega(n \lg n)$ or a tighter $T(n) = \Theta(n \lg n)$. The best case maps to the case where the splitter is the median of the n input keys (initially) or the participating keys in every recursive call.*

Proof. We can work similarly to the worst case. The only change will be finding not the maximum of $f(m) = m^2 + (n - m - 1)^2$ but the MINIMUM of function $f(m) = m \lg m + (n - m - 1) \lg(n - m - 1)$. The latter has a minimum at $i \approx n/2$, which shows that the best case performance is attained for a 50 - 50 split. ■

Exercise 2. Show similarly to the $O(n^2)$ case, that for the best-case $T(n) = \Theta(n \lg n)$, by showing first $T(n) = \Omega(n \lg n)$, and then $T(n) = O(n \lg n)$.

Problem Set 1 might help as well (entropy function problem).

Quicksort

GTQuickSort average-case performance analysis

Theorem 3. *The average case running time of GTQuickSort can be shown to be $T(n) = \Omega(n \lg n)$ or a tighter bound $T(n) = \Theta(n \lg n)$.*

Proof. For the average case performance we use that all splits are equally likely and average the possibilities.

$$T_{ave}(n) = \frac{1}{n} \left\{ \sum_{m=0}^{n-1} T_{ave}(m) + T_{ave}(n - m - 1) + \Theta(n) \right\}$$

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. (A bad split is at worst a 0 and $n - 1$ split, a good split is at best a 50-50 split.) On the average a bad split is followed by a good split: the worst split can be at worst a 0 and $n - 1$, the subsequent good split is $(n - 1)/2$ and $(n - 1)/2$, i.e. we have a 50-50 split AFTER EVERY TWO iterations, on the average. This leads to an argument that the recursion tree has expected depth $2 \lg n$ and thus average case performance is twice as bad as best case performance but asymptotically equal and equal to $\Theta(n \lg n)$. ■

Note: We shall drop the subscripts from the T symbol in the remainder to keep things simple and the exposition compact. The subscript however would be implied by the case analyzed (best vs worst vs etc.)

1. What if we have 25%-75% or 33%-66% split?

If instead of an $n/2$ vs $n/2$ split we have $n/4$ and $3n/4$ split in every iteration, the recursion is still $\Theta(\lg n)$ deep. The recurrence is of the form $T(n) = T(n/4) + T(3n/4) + \Theta(n)$. A substitution method invocation confirms that $T(n) = \Theta(n \lg n)$. The running time is asymptotically the same for any constant $c > 2$ based split i.e. $T(n) = T(n/c) + T((c-1)n/c) + \Theta(n)$. The constant hidden in the $\Theta(\cdot)$ however depends on the difference in size of the left side and the right side and grows with a growing difference.

2. GTQuicksort properties.

QuickSort and GTQuicksort does not sort in-place. One may claim that this due to its recursive nature, but we can have an iterative version of QuickSort. It is because of the depth of the recursion of a recursive or iterative approach. In the best case it is $\Theta(\lg n)$ in the worst case it is $\Theta(n)$. Thus the stack of recursive calls is not constant $\Theta(1)$ but $\Omega(\lg n)$. In the stack we store pairs (l, r) or $(l, m-1)$ and $(m+1, r)$, as needed. Moreover GTQuicksort is not stable. It is left as an exercise to provide an input that proves the lack of stability of GTQuicksort.

3. QuickSort and its running time.

The running time of GTQuicksort is derived from its worst-case running time. The latter is $W_{\text{QuickSort}}(n) = \Theta(n^2)$, according to previous derivations. Moreover its best case running time is $B_{\text{QuickSort}}(n) = \Theta(n \lg n)$. slides. Thus following prior discussion we derive that the running time is $T(n) = T_{\text{GTQuickSort}}(n) = O(n^2)$.

Quicksort

GTQuickSort : conclusion

Proposition 2 (GTQuickSort). *GTQuickSort is a divide and conquer sorting algorithm whose performance can be summarized as follows. Its running time is $T(n) = O(n^2)$, since it exhibits best case performance of running time $T(n) = \Theta(n \lg n)$. GTQuickSort is not stable (page 9) neither does it sort in-place.*

DRAFT

Randomized Quicksort

Introduction

A **randomized algorithm** is one whose behavior is also determined by the values of a random number generator (rng). This way bad inputs don't necessarily lead to worst-case running time; the rng might smoothe out the bad behavior of bad inputs. Good inputs might not be good inputs any more: the rng might roughen up the apparent good behavior of good inputs. The overall effect of the rng is to average out: make bad inputs behave better, and good input behave worse but not worst. Running time then depends on the choices and values of the rng. However the chances of an rng generating bad choices, even if it is NOT ZERO, it is very small.

The chances of a rng generating bad choices AGAIN and AGAIN are even smaller. And even if a worst-case occurs because of the rng's current behavior we always have a quick fix. Stop the current execution, rerun the (randomized) algorithm and hope that the second time around we would be luckier with the rng's choices! Randomized QuickSort will be denoted by **GTRQuickSort**. It is a version of QuickSort where the splitter is picked to be a random input key. To generate **GTRQuickSort** we only need to perform minor changes to GTQuickSort (in fact GT). This means replacing GTPartition with a GTRPartition. We follow a code reuse approach of software engineering that does not require a rewrite of GTPartition. In short, before we call GTPartition, we first pick a random (in practice pseudo random) integer in the range $l..r$ and we use that integer, say i , to generate a random splitter $A[i]$. Since GTPartition uses $A[r]$ as splitter, we move $A[i]$ to index r i.e. we swap $A[i]$ and $A[r]$. Code follows. It shares all properties of GTQuickSort including running time $T(n) = O(n^2)$.

Randomized Quicksort

GTRQuickSort

Method 5 (GTRQuickSort). The code for GTRQuickSort that implements a randomized version of quicksort utilizing GTRPartition is shown below. The pivot key is a random input key that moves to the rightmost key of the input sequence before GTPartition takes over from GTRPartition.

```
GTRQuickSort (A[0..n-1],n)
1.  GTR(A,0,n-1);

    GTR(A,l,r)  // A[l..r]
1.  if ( l < r ) {
2.      m= GTRPartition(A,l,r); // A[l,...,m-1,m,m+1,...,r]
3.      GTR (A,  l, m-1);        // sort L = A[l .. m-1]
4.      GTR (A, m+1, r );        // sort R = A[m+1 .. r ]

    GTRPartition (A,l,r)
1.  i = RandomInteger(l,r)      // pick a 'random' int i in range l..r
2.  swap( A[i], A[r]);          // 'random' splitter moves from i to r
3.  return( GTPartition(A,l,r)); // GTPartition takes over for the rest
```

(An overview is on the next page.)

Randomized Quicksort

GTRQuickSort overview

Overview of GTRQuickSort. **GTRQuickSort on fixed input S , a sorted sequence.** Let S be a sorted sequence of n integer keys $1, \dots, n$ and V be a reverse sorted sequence of those n keys. We know that **GTQuickSort**(S, n) exhibits worst case performance on input S or V that is quadratic and thus has running time $T(n) = \Theta(n^2)$. For GTRQuickSort let its input also be S . Even if the input is a sorted sequences, GTRQuickSort picks a random splitter that moves into the position occupied by the maximum (largest input) key. Thus in the first iteration it is unlikely to observe worst-case performance; likewise in the recursive calls that follow. Thus randomization removes the burden of bad 'inputs'. There is no bad input(s) anymore, they are smoothed out by the randomly chosen splitter. Can worst-case performance occur? Not only yes it can, but it can occur in a sorted sequences such as S , in a reverse sorted sequence such as V and in fact in any input sequence, i.e. in any of the $n!$ permutations of an arbitrary set of n input keys. The probability that this occurs however is very very low.

Why? Think about it. What are the chances the random splitter in the first iteration is the maximum? It is $1/n$. This is the same whether the input is S or V or any other of the $n!$ permutations of the input. Say the splitter was the maximum; we then recur on the set containing $n - 1$ keys. What are the chances that we pick that set's maximum as the splitter again? $1/(n - 1)$ is the probability now. And so on and so on we realize that the probability of generating a worst-case scenario from any input is $1/n!$. (The catch: it is twice as much as the same argument can be applied to the minimum and this doubles the probability of worst-case performance.) However $1/n!$ or $2/n!$ for GTRQuickSort on input S (a sorted sequence) is much less than the probability 1 (always worst-case performance) for QuickSort on S . In the deterministic case when the input is a sorted sequence, we always observe worst-case performance. In the randomized case, we observe such performance very rarely $2/n!$. **(We do not eliminate worst-case performance, we make it rare.)** Algorithm GTRQuickSort has interesting performance. The performance of GTRQuickSort is independent of the input. It only depends on the choices (or properties) of the random number generator. There is no bad input anymore. There are only bad choices of the random number generator.

Even on the same fixed input, running time can vary depending on the choices of the random number generator that generates the random (or pseudo random) i . Thus the running time is a random variable. The following theorem describes the expected value of that random variable.

Thus, the running time $T(n)$ becomes a random variable and it makes sense to discuss the expectation of this random variable $\tilde{T}(n) = E[T(n)]$. We shall prove that this is $\Theta(n \lg n)$. ■

Theorem 4. *The expected running time of GTRQuickSort is $\tilde{T}(n) = E[T(n)] = \Theta(n \lg n)$.*

Proof. With reference to any input we call **bad splitters** the ones that have behavior similar to the minimum or the maximum. Everybody else is a **good splitter**. To simplify the exposition assume all the keys are integer 1 through n , or consider those values to be the ranks of the input keys (rank 1 is smallest/minimum, rank 2 second smallest, rank $n - 1$ second largest, rank n largest/maximum). Thus we use notation **BadSplitter** and **GoodSplitter** from now on. The **BadSplitter** are the inputs $1..n/4$ and $3n/4..n$. In general, they are the quarter of the smallest and the quarter of the largest keys. The **GoodSplitter** are the rest i.e. $n/4 + 1..3n/4 - 1$. (Assume n is a multiple of 4 to avoid floor and ceilings.) If GTRPartition encounters a GoodSplitter, the split in the worst case is 25% - 75% up to the best possible 50% - 50% . All such cases are treated the same and we assign to such cases a running time bound provided by the (worst best-case split of) a 25% - 75 % split not the best one. However prior comments indicate that GoodSplitter leads to $\Theta(n \lg n)$ time. How about a BadSplitter pick as a splitter? We treat it as 0 and $n - 1$ split even if $n/4$ was picked as the splitter and thus the split is slight worse than 25% - 75 % say a 24% - 76 % . Thus all such split are assigned a $\Theta(n^2)$ cost even if some are not much worse than a GoodSplitter. Then we use a prior argument claim that on the average half of the choices of a random splitter are GoodSplitter and half are BadSplitter. In two iterations of Bad and Good a 25% - 75 % split results. The running time will $\Theta(n \lg n)$ and it is the expected running time of GTRQuickSort. **The depth of the recursion is going to be twice as bad as the depth of a 25% - 75 % split which by itself is worse than that of a 50% -50% split. But all those constants are inside a $\Theta(\cdot)$: the power of asymptotics.** Case proven. If we denote with $\tilde{T}(n)$ the expected running time of RQuickSort we get

$$\tilde{T}(n) = \Theta(n \lg n).$$

■

Randomized Quicksort

GTRQuickSort alternate performance analysis

Theorem 5 (Alternative). *The expected running time of GTRQuickSort is $\tilde{T}(n) = E[T(n)] = \Theta(n \lg n)$.*

Proof. If you are not convinced by the “average” arguments of the previous slide, then the alternative is a formal approach. Say the splitter is the largest of the n keys ($m = n - 1$). 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, $\dots$, $n - 2$ with the same probability $1/n$. Therefore the left set is of size 0, 1, 2, 3, $\dots$, $n - 1$ with probability $1/n$.

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

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

Then the recurrence of the running time is given below.

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

Assumption. To keep things simple the last term is n i.e. $1 \cdot n$. You can think of this term as the comparison contribution of GTRPartition. One might argue for $n - 1$ or $2n$ or $2n - 2$ or in general $c \cdot n$ where c is some positive integer constant. Redo the calculation of the next two slides and see what happens then!

So far we have shown that,

$$T(n) = \frac{1}{n} \left(\sum_{i=0}^{n-1} T(i) + T(n - i - 1) \right) + n \tag{4}$$

We prove the result by solving this recurrence with the substitution method. (A solution using the iteration method follows.) ■

GTRQuickSort alternate performance analysis continued

Analytical solution. We multiply Eq 4 by n both sides and then substitute in the resulting equality $n - 1$ for n . We write both derivations below

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

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

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + n^2 - (n-1)^2 \iff nT(n) - (n-1)T(n-1) = 2T(n-1) + (2n-1) \iff nT(n) = (n+1)T(n-1) + (2n-1) \iff \\ \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{(2n-1)}{n(n+1)} \end{aligned}$$

Now let us substitute $t(n) = T(n)/(n+1)$. We get

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{(2n-1)}{n(n+1)} \iff t(n) = t(n-1) + \frac{(2n-1)}{n(n+1)}$$

By Observation 1 we have $t(0) = t(1) = 0$. We then upper bound the last term. An alternative method is to expand the last term as in $(2n-1)/(n(n+1)) = -1/n + 3/(n+1)$.

$$t(n) = t(n-1) + \frac{(2n-1)}{n(n+1)} \iff t(n) = t(n-1) + \frac{-1}{n} + \frac{3}{n+1}$$

Expanding the recurrence (iteration method) we get

$$\begin{aligned} t(n) &= t(n-1) + \frac{-1}{n} + \frac{3}{n+1} \iff t(n) = t(1) - (1/2 + \dots + 1/n) + 3(1/2 + \dots + 1/(n+1)) \iff t(n) = -(1/2 + \dots + 1/n) - 3 + 3(1 + 1/2 + \dots + 1/n + 1/(n+1)) \\ t(n) &= 2(1 + 1/2 + \dots + 1/n) + 3/(n+1) - 3 \iff t(n) = 2H_n + \frac{3}{n+1} - 3 \text{ if } T(n)/(n+1) = 2H_n + 3/(n+1) - 3 \\ T(n) &= 2(n+1)H_n + 3 - 3(n+1) \\ T(n) &\leq 2n \ln n \end{aligned}$$

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 $T(n) \approx 2n \ln n$. ■

Complexity of comparison-based sorting

Lower bounds

Let for comparison-based sorting

$$Sort(n) = \min_A T_A(n)$$

This means find the best algorithm A with the best worst-case running time i.e. best algorithm A with the best running time.

Theorem 6. *We know that*

$$Sort(n) = \Theta(n \lg n).$$

Proof. Algorithms such as SelectionSort and Odd-EvenTranspositionSort have $T(n) = \Theta(n^2)$.

Algorithms such as GTQuickSort and InsertionSort have $T(n) = O(n^2)$.

Algorithms such as HeapSort and MergeSort have $T(n) = \Theta(n \lg n)$.

Algorithms such as NaiveSort have $T(n) = O(n!n)$.

The best algorithms with the best running time is a choice of MergeSort and HeapSort. ■

Question 2. Is there a sorting algorithm X that has worst-case performance (running time) better than $\Theta(n \lg n)$?

Complexity of comparison-based sorting

Lower bounds

Let for comparison-based sorting

$$SORT(n) = \min_A C_A(n)$$

$C_A(n)$ is the comparison-count of sorting algorithm A . This means find the best algorithm A with the best worst-case comparison count.

Theorem 7. *We know that*

$$SORT(n) \leq n \lg n - n + 1$$

Proof. Algorithms such as SelectionSort and InsertionSort have worst-case comparison count $C(n) = n(n-1)/2$.

Algorithms such as HeapSort have $C(n) \leq 2n \lg n + 2n$.

Algorithms such as MergeSort have $C(n) = n \lg n - (n-1)$.

The best algorithm with the best worst-case comparison count is MergeSort. ■

Question 3. Is there a sorting algorithm X that has worst-case comparison count smaller than the $n \lg n - n + 1$ of MergeSort?

Theorem 8. *We shall show that any comparison-based sorting algorithm X cannot perform fewer than $\lceil \lg(n!) \rceil$ comparisons in the worst-case. Therefore,*

$$SORT(n) \geq \lceil \lg(n!) \rceil.$$

The $\lceil \lg(n!) \rceil \approx n \lg n - n \lg e \approx n \lg n - 1.44n$. Thus there is the possibility that there exists a sorting algorithm with worst case comparison count $n \lg n - an$ where a is a constant such that $1 < a < \lg e \approx 1.44$.

Complexity of comparison-based sorting

Lower bounds

Question 4. For $n = 2$ the lower bound says $SORT(2) \geq \lceil \lg(2!) \rceil$. Is this bound tight i.e. does there exist a sorting algorithm whose worst-case comparison bound matches the lower bound?

Proof. YES. $\lceil \lg(2!) \rceil = 1$. We can sort 2 keys with one comparison that determines MIN and MAX. ■

Question 5. For $n = 3$ the lower bound says $SORT(3) \geq \lceil \lg(3!) \rceil$. Is this bound tight i.e. does there exist a sorting algorithm whose worst-case comparison bound matches the lower bound?

Proof. YES. $\lceil \lg(3!) \rceil = 3$ and InsertionSort uses $n(n-1)/2$ which is 3 for $n = 3$. ■

Question 6. For $n = 4$ the lower bound says $SORT(4) \geq \lceil \lg(4!) \rceil$. Is this bound tight i.e. does there exist a sorting algorithm whose worst-case comparison bound matches the lower bound?

Proof. YES. $\lceil \lg(4!) \rceil = 5$ and InsertionSort uses on the first three keys uses 3 of the 5 comparisons. We then do BinarySearch Insertion of the fourth key into the sorted sequence of the first three keys. This can be done in two comparisons for a comparison total of five as needed (for the worst-case). ■

Question 7. For $n = 5$ the lower bound says $SORT(5) \geq \lceil \lg(5!) \rceil$. Is this bound tight i.e. does there exist a sorting algorithm whose worst-case comparison bound matches the lower bound?

Proof. YES. Some merging might be needed. It get trickier with increasing n . ■

Complexity of comparison-based sorting

Lower bounds

Theorem 9. *A binary tree with m leaves must have height at least $\lg m$.*

Proof. Say height of such a tree with m leaves is h such that $h < \lg m$.

A binary tree with height 0 has number of leaves $\leq 2^0$.

A binary tree with height 1 has number of leaves $\leq 2^1$.

And thus, a binary tree with height h has number of leaves $\leq 2^h$.

Since $h < \lg m$ this means that the height h tree has fewer than $< 2^{\lg m} = m$ leaves. This contradicts to it having m leaves. Thus $h \geq \lg m$. ■

DRAFT

Complexity of comparison-based sorting

Lower bounds

In a comparison based sorting algorithm we assume that the input keys are comparable that is, a total order is defined on the input keys.

Whenever two keys are to be compared, a comparison function `cmp` is to be used. The result of the comparison between two keys x and y is indicated by `cmp(x,y)` and is (a) a positive integer or +1 to indicate $x > y$, (b) a zero to indicate that x is equal to y , and (c) a negative integer or -1 to indicate $x < y$. Several times we represent a comparison with a colon $:$. Thus $x : y$ is the same as `cmp(x,y)`.

Definition 1 (*Decision tree model*). A **decision tree** is usually a full binary tree that represents the comparisons performed between objects on which a total order is defined during the execution of an algorithm that solves a related problem. (For sorting the objects are the input keys.) Control, data exchanges and other details of the algorithm are ignored. For an algorithm whose input consists of n comparable objects $a_1, \dots, a_n$, the decision tree is a binary tree where each internal node contains a comparison between two keys a_i, a_j i.e. $a_i : a_j$ for some $i \neq j \leq n$. A leaf maps to an outcome of the algorithm.

Definition 2 (*Decision tree model for sorting*). For a sorting algorithm whose input consists of n comparable keys $a_1, \dots, a_n$, the decision tree is a binary tree where each internal node contains a comparison between two keys a_i, a_j i.e. $a_i : a_j$ for some $i \neq j \leq n$. For sorting we assume that the input keys to a decision tree are distinct. Thus in an internal node a $a_i : a_j$ comparison can have only two outcomes $a_i < a_j$ and the left branch is followed or $a_i > a_j$ and the right branch is followed. A leaf maps to an outcome of the algorithm that is, a permutation of the input keys $\langle a_{j_1}, \dots, a_{j_n} \rangle$ that is compatible with sorting i.e. $a_{j_1} \leq \dots \leq a_{j_n}$.

If $=$ is allowed in the input the two branches of an internal node can be interpreted for example as $a_i \leq a_j$ and $a_i > a_j$. Several times a decision tree might become a ternary tree thus allowing for $a_i < a_j$, $a_i > a_j$ and $a_i = a_j$.

Complexity of comparison-based sorting

Decision trees

Definition 3 (*Decision tree for sorting root to leaf path*). An execution of a sorting algorithm maps to a path from the root that contains the input to a leaf that maps (contains) the corresponding output. The path shows the comparisons that were performed between any two keys. The length of the path (in edges) is the number of comparisons performed in that execution. If at some node containing the comparison $a_i : a_j$, a left path was taken this means that $a_i < a_j$ was decided, otherwise $a_i > a_j$.

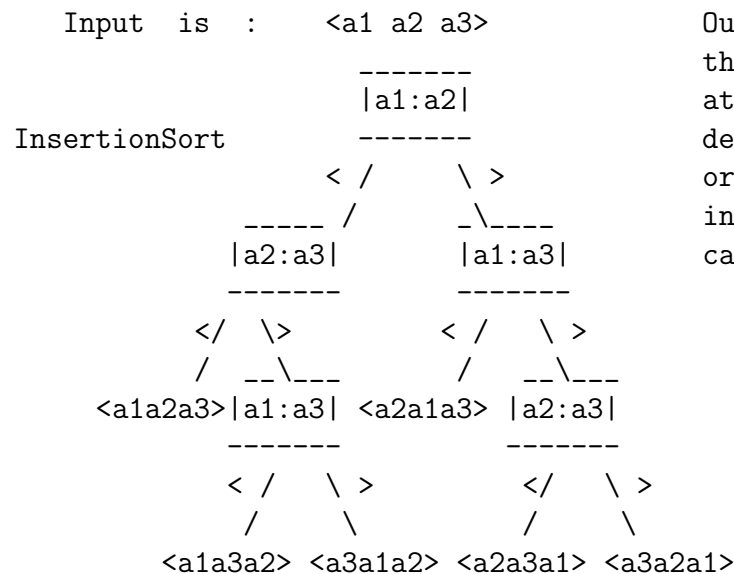
Definition 4 (*Reaching a leaf*). When a leaf is reached, this means that the output sequence has been sorted and the sorting algorithm has established the appropriate ordering of the input keys, $\langle a_{j_1}, \dots, a_{j_n} \rangle$ that is compatible with sorting i.e. $a_{j_1} \leq \dots \leq a_{j_n}$. As the number of permutations on n (distinct) input keys is $n!$ and each permutation must appear as a leaf (provided that the sorting algorithm is correct i.e. it DOES sort any permutation of n distinct keys) in the decision tree, we conclude that any decision tree for sorting SHOULD have at least $n!$ leaves.

Definition 5 (*Height of a decision tree for sorting*). A path from the root to a leaf shows one execution and its length the cost of the running time for a specific input. The worst case running time of a sorting algorithm on n keys, is the height of its decision tree for sorting n (distinct) keys.

Decision tree A three-key decision tree for InsertionSort

Example 2 (InsertionSort). We show the decision tree for InsertionSort where the input consists of 3 distinct keys $\langle a_1, a_2, a_3 \rangle$.

Proof.



Output is a leaf and shows the input sequence sorted at the corresponding leaf depending on the relative order of the input key values in each one of 3! possible cases.

$$\begin{array}{rcl} \text{Average \# of comparisons} & & \\ (2+3+3+2+3+3) & 16 & \\ \hline 3! & = & \frac{16}{6} = 2.5+ \end{array}$$

A three-key decision tree for InsertionSort

Observations

Some observations. The leaves show the keys without separating spaces. All $3!=6$ permutations of the input sequence appear as leaves. If $\langle a_1, a_2, a_3 \rangle$ is the reached leaf it means that on input $\langle a_1, a_2, a_3 \rangle$ InsertionSort generates output $\langle a_1, a_2, a_3 \rangle$ and we know that the output is such that $a_1 \leq a_2 \leq a_3$. This means not only the output is sorted but so is the input. Likewise leaf $\langle a_3, a_2, a_1 \rangle$ indicates $a_3 \leq a_2 \leq a_1$ i.e. input is reverse sorted and thus $a_1 \geq a_2 \geq a_3$.

Theorem 10 (Minimum height of decision tree for sorting). *The minimum height is $n - 1$.*

Proof. Say the height is less than $n - 1$. Consider a graph with n vertices each one being an input key a_1, a_2, a_3 , etc. Every comparison in the execution that involves $a(i) : a(j)$ generates an edge $(a(i), a(j))$ in the graph. The graph must be connected i.e. it must be a tree and has $n - 1$ edges!

Why ? If it is not connected, pick a vertex in one connected component say $a(k)$ and another vertex in the other connected component $a(m)$. Those two vertices were never compared nor can we say indirectly (through transitivity) which one is smaller or larger. Then the leaf of a path $< n - 1$ cannot uniquely indicate which key is smaller or larger: another way to say so is to arrange keys $a(i)$ and $a(k)$ so that the indicated leaf is not a sorted sequence. ■

As a path from the root to a leaf gives a computation trace of the algorithm for some input, the length of the longest path represents the worst case number of comparison steps required to sort an input sequence of n keys. (In addition the leaf provides us with information about the input.)

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.

Decision tree Adversarial argument setup

Another way to view a decision tree is from the point of view of an adversary B ("Bad guy") who plays a game with an "Algorithm designer" person A. Person A is challenged by B. B claims that A's sorting algorithm in the worst case takes at least $\lceil \lg(n!) \rceil$ number of comparisons and A is not convinced about this. So B will form a sequence of n (distinct) keys (in our example $n = 3$) that will guarantee a comparison count of at least $\lceil \lg(n!) \rceil$ when A's algorithm is applied. B will not initialize the keys' value in advance: he will make decisions about the key values while A runs the algorithm on B's not yet established input. So B will make a decision about a key comparison say $a_i : a_j$ depending on whether it leaves B with more freedom of choices later on: in fact whether size C of possible orderings consistent with the answer to a question is as large as possible. A will simulate (i.e. compare keys) as if InsertionSort is running.

```

Input      : a1a2a3  [Formed by B but B has not decided on the ordering of keys]
Possible Orderings : C = {a1a2a3, a1a3a2, a2a1a3, a2a3a1, a3a1a2, a3a2a1}.
                  |C| = 3! = 6
Note       : Assume all keys distinct

InsertionSort
    |a1:a2|
    -----
    < /      \ >
C={a1a2a3,a1a3a2,a3a1a2} /      \ C={a2a1a3,a2a3a1,a3a2a1}
    ----- /      \ -----
    |a2:a3|      |a1:a3|      Comp 2:
    -----      -----
    < /      \ >
C={a1a2a3}    /      \ C={a1a3a2,a3a1a2}
    /      \ -----
    /      \ |a1:a3|
                -----
                < /      \ > C={a3a1a2}
                /      \
                a1a3a2    a3a1a2

Comp 1:  A asks B : compare(a1,a2)
        < leads to a new C of |C|=3!/2
        > leads to a new C of |C|=3!/2

        B can pick < or > ; B picks <

Comp 2:  A asks B : compare(a2,a3)
        > leads to a new C of |C|=2
        since < leads to a new C of |C|=1
        B picks >

Comp 3:  A asks B : compare(a1,a3)
        B picks < and comments as follows:
        " I, B, forced you A to perform
        ceil(lg(3!))=3 comparisons on
        my input of choice which is
        a1 = 1 , a3 = 2 , a2 = 3".
        [B could have picked > for a1:a3]
        A can verify B is telling the truth.

```

■

Theorem 11. *Any decision tree that sorts n keys (using comparisons) has height at least $\lg(n!)$ or asymptotically $\Omega(n \lg n)$.*

Proof. A decision tree that sorts n keys must have at least $n!$ leaves, since n (distinct) keys generate $n!$ permutations and each such permutation is sorted by following a path to one of the $n!$ leaves. (A binary tree of height h has at most 2^h leaves; another way to say that is the shallowest tree with 2^h leaves is the complete binary tree of height h .) Therefore a binary tree (the decision tree) of $n!$ leaves has height at least $\lceil \lg(n!) \rceil$. Therefore we need that $2^h \geq n!$ ie $h \geq \lg(n!)$. From Stirling's formula for the factorial, we get $n! \geq (n/e)^n$. This implies that $h \geq n \lg n - n \lg e = \Omega(n \lg n)$. ■

Corollary 1. If the decision tree has height at least $\lg(n!)$ it means the worst-case comparison count of any comparison-based sorting algorithm is at least $\lg(n!)$ and can be no less (in the worst-case).

Corollary 2. Heapsort and mergesort are asymptotically optimal but not necessarily worst-case comparison count optimal.

It is possible that a sorting algorithm exists that has comparison count in the worst case better than that of MergeSort i.e. say $C(n) = n \lg n - 1.2n$.

Exercise 3. Can we sort 5 arbitrary keys with at most 7 comparisons in the worst-case? How about 6? Explain.

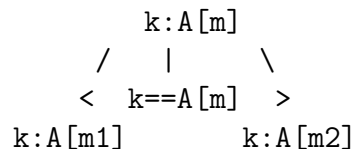
Decision tree Binary search

Example 3. For a sorted input of n comparable keys in array A , $\text{BinarySearch}(A, n, k)$ returns either an index $0 \dots n - 1$ to indicate that $A[i] = k$ or a -1 for a NOT-FOUND condition. There are a total of $n + 1$ possible outcomes. One can use a decision tree argument to show that searching with comparisons (in binary search) takes at least $\lceil \lg n \rceil + 1 = \lceil \lg(n + 1) \rceil$ comparisons in the worst case.

We demonstrate this through an example of a 7-key A and thus $n = 7$, $A = \langle x_0, x_1, x_2, x_3, x_4, x_5, x_6 \rangle$ and a key k .

■

Proof. Proof. A decision tree for binary search must be such that at every node we examine one position of A . The tree must have at least n (internal) nodes (one for each position of A) and thus the number of external nodes (leaves) must be at least $n + 1$. This leads again to the $\lceil \lg(n + 1) \rceil$ bound. Consider a decision tree of height $h - 1$. Every node looks like the one below. By comparing k and $A[m]$ we have three branches to the next level for the $<$, $=$, and $>$ cases. The height of the example is 1 and in the worst-case we perform 2 comparisons! Thus for a height $h - 1$ decision tree we would perform in the worst case h comparisons. The number of levels in such a tree is h from 0 through $h - 1$. The number of comparison nodes must be at least n (one per $A[i]$). The number of such nodes at level 0 is at least 2^0 at level 1 is at least 2^1 and likewise at level $h - 1$ is at least 2^{h-1} for a total of $2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1$. For this to be at least n we must have $2^h - 1 \geq n$ i.e. $h \geq \lceil \lg(n + 1) \rceil$.



■

Thus $h \geq \lceil \lg(n + 1) \rceil$ is the worst-case comparison count in BinarySearch .

Question 8. If you use a scale to weigh a collection of coins can you spot the fake coin? How many weighings minimally do you need ? (a) 27 coins, one fake and lighter, (b) 3 coins one is fake but don't know if lighter or not!

DRAFT

Sorting in linear time

Review

The sorting algorithms we have examined so far are **comparison** and **exchange (swap)** 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 get swapped (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 fact we show that the number of comparisons needed in the worst case is at least $\lg(n!)$ thus implying an $\Omega(n \lg n)$ running time bound. In order to prove this bound we shall introduce a comparison metric structure that is a binary tree and is known as 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.

A third algorithm is also going to be introduced. The three algorithms are listed below.

Problem 1 (CountSort, BinSort, BucketSort).

Input: The input keys are integers in the range $0, \dots, k - 1$. There are n input keys.

Output: Sort the n input keys in time that is $\Theta(n + k)$.

Sometimes the range is $1, \dots, k$. The name BucketSort is being used when linked lists are maintained during sorting. If arrays are maintained only then the name utilized is CountSort. The name BinSort is an alternative name for one or the other setup.

The term radix indicates the base of an integer. Thus “denary” (also referred to as decimal) are radix-10 integers. Binary are radix-2.

Problem 2 (RadixSort).

Input: The input keys are integers in the range $0, \dots, k^d - 1$. There are n input keys.

Output: Sort the n input keys in time that is $\Theta(d(n + k))$. This is realized by d rounds of CountSort. The key values are viewed as forming a d -digit number where the radix of a digit is k (thus allowing for values $0, \dots, k - 1$).

Example 4. Suppose we have n integers in the range $0, \dots, 10^6 - 1$. We may view those n integers in the traditional way as 6-digit radix-10 integers. Then, $k = 10$ and $d = 6$.

Problem 3 (Probabilistic Sort).

Input: The input keys are real numbers in the range that is the closed interval $[a, b]$ and uniformly at random distributed in it. There are n input keys.

Output: Sort the n input keys in expected time that is $\Theta(n)$.

Example 5. Suppose we have n integers that are of the C datatype `unsigned int`. They are non-negative integers in the range $0, \dots, 2^{32} - 1$. We may view those n integers in the following ways.

- 32-digit radix-2 integers that is $d = 32$ and $k = 2$.
- 4-digit radix-256 integers that is $d = 4$ and $k = 256$. (In other words a byte becomes a digit.)
- 2-digit radix-65536 integers that is $d = 2$ and $k = 65536$. (In other words a half-word that is two consecutive bytes become a digit.)
- 1-digit radix- 2^{32} integers that is $d = 1$ and $k = 2^{32}$. (In other words all four bytes form a single digit.)

Therefore integer `x= 00000001 00000000 10000000 00000011` can be viewed as

- (i) 32-digit radix-2 integer $\langle 0, 0, 0, 0, 0, \dots, 0, 1, 1 \rangle$, or
- (ii) 4-digit radix-256 integer $\langle 1, 0, 128, 3 \rangle$, or
- (iii) 2-digit radix-65536 integer $\langle 256, 32771 \rangle$.

Example 6. Let $x = 1234$ be a 4-digit radix-10 integer. We erroneously call it decimal instead of denary. For x we have $d = 4$ and $k = 10$. Such 4-digit numbers are

$$x \leq k^d - 1 = 10^4 - 1 = 9999$$

Example 7. Let $y = 01001101$ be a 8-digit (8-bit) radix-2 integer. We call it a binary number. For x we have $d = 8$ and $k = 2$. Such 8-bit numbers are

$$x \leq k^d - 1 = 2^8 - 1 = 255.$$

Example 8. With $\lg k$ bit we can represent $2^{\lg k} = k$ values e.g. $0 \dots k - 1$.

With $2 \lg k$ bit we can represent $2^{2 \lg k} = k^2$ values e.g. $0 \dots k^2 - 1$.

With $d \lg k$ bit we can represent $2^{d \lg k} = k^d$ values e.g. $0 \dots k^d - 1$.

Distribution-based sorting

Countsort

Let us assume that n keys to be sorted take values from a set $0, \dots, k - 1$. We are going to present an algorithm whose running time is $O(n + k)$. Then IF $k = O(n)$ this is $\Theta(n)$, i.e. 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.

DRAFT

Distribution-based sorting

Linked list-based Bucketsort

Theorem 12 (BucketSort). *Sorting n integer keys in the range $0, \dots, k - 1$ can be done in time $\Theta(n + k)$ using algorithm BucketSort. BucketSort is stable and does not sort in-place.*

Proof. In the BucketSort variant a bucket or a bin is a linked list. Thus bucket i or bin i stores all the keys whose values are i . With a linear scan left to right, each input key can be placed (inserted) into the appropriate bin using a linked list Insert operation. We only need to concatenate the lists at the end. For stability Insert inserts at the end of the list. An alternative is to insert at the head of the list but scan the input keys right to left. The algorithm uses a different array for input and output thus it does not sort in-place. The running time of the algorithm is $\Theta(n + k)$.

```
BucketSort(I, O, n, k) // I[0..n-1] is input  O[0..n-1] is output
1. for(i=0; i<k; i++)
2.   InitializeListfor(Bucket[i]); // Create k NULL buckets
   // Bucket[i].len
3. for(j=0; j<n; j++)
4.   InsertList(A[j], Bucket[A[j]]);
5. Initialize(O, n) // Make sure output O is empty
6. count=0;
6. for(i=0; i<k; i++)
7.   Copy(O, count, Bucket[i].length); //O[count ...] = Bucket[i]
8.   count = count + Bucket[i].length;
9. return(O);
```

■

Linked list-based Bucketsort Example

Example 9.

0 1 2 3 4 5 6 7 n = 8 k = 4

I : < 1 0 2 0 3 1 0 3 >

L0 -> o L0-> 0<->0<->0

L1 -> o READ L1-> 1<->1 CONCATENATE 0 0 0 1 1 2 3 3

L2 -> o INPUT L2-> 2 LISTS

L3 -> o L3-> 3<->3 L0-L1-L2-L3

Question 1? 0 at I[1] is it the leftmost or rightmost 0 of L0?

Question 2? Stable?

Question 3? IN-PLACE ? DEFINITELY NOT.

Distribution-based sorting

Array-based Countsort

Theorem 13 (CountSort). *Sorting n integer keys in the range $0, \dots, k - 1$ can be done in time $\Theta(n + k)$ using algorithm CountSort. CountSort is stable and does not sort in-place.*

Proof.

```
CountSort(I, 0, n, k)
1. for(i=0; i<k; i++)
2.   C[i]=0;           // Initialize the Counter Array of length 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[m] : number of keys with value m
5. for(i=1; i<k; i++)  // C[i] becomes #keys with values at most i.
6.   C[i] = C[i]+C[i-1]; // C[i]-1 is the last index of 0 holding an i
7. for(j=n-1; j>=0; j--) {
8.   C[I[j]]--;
9.   O[C[I[j]]]=I[j];
   }
```

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

The bound of CountSort does not contradict the lower bound for sorting: CountSort is not a comparison and exchange sorting algorithm.

Note also that it is a **stable** algorithm, i.e. equal keys in the input sequence preserve their relative order in the output sequence; we need to fill the output right-to-left to accomplish that for the given structure of C . ■

Definition 6 (Prefix sums). For the sequence of values a_1, a_2, a_3 the prefix sums of the sequence is sequence $a_1, a_1 + a_2, a_1 + a_2 + a_3$. Lines 5-6 of CountSort compute the prefix sums of $C[0], C[1], \dots, C[k - 1]$.

Array-based Countsort Example

Example 10.

```

      0 1 2 3 4 5 6 7
I : [ 1 0 2 0 3 1 0 3 ]   n=8 k=4

Steps 1,2  Steps 3,4      P
C 0: 0      C 0: 3 | | | |r S  C 0: 3  In output      0: is a 2
  1: 0      1: 2  | | | |e u    1: 5  right most      1: is a 4
  2: 0      2: 1  | |f m      2: 6  index storing a 2: is a 5
  3: 0      3: 2  |i s      3: 8                      3: is a 7
                        x          ^                      ^
                        ----- C[]-1 is -----

```



```

      0 1 2 3 4 5 6 7
O : [ 0 0 0 1 1 2 3 3 ]

```

Array-based Countsort Example

Example 11. An example of CountSort is shown. It is a continuation of the previous one.

Proof. An example follows.

```

CountSort([1,0,2,0,3,1,0,3], [], 8, 4)
      \   |
1.      \   |
2. C=[0,0,0,0]      \   |
      \   |
3.      |   | No crossing
4. C=[3,2,1,2]      |   | for the 3s
      |   |
5.      |   | How about
6. C=[3,5,6,8]      |   | the 0,1,2?
   C[i] last index  |   |
7.   position occupied |   |
8.   by i              |   |
9.   0=[0,0,0,1,1,2,3,3]

```

DRAFT

■

Remark 1. CountSort has performance that is linear to n and linear to k . This does not necessarily mean that its performance is linear to problem size, which is n . Consider the case where $k = n^2$. The term $\Theta(n + k)$ is $\Theta(n + n^2) = \Theta(n^2)$.

Theorem 14 (RadixSort). *Sorting n integer keys in the range $0, \dots, k^d - 1$ can be done in time $\Theta(d(n + k))$ using algorithm RadixSort. Algorithm RadixSort uses d rounds of algorithm CountSort. Each key value is viewed as a d -digit radix- k integer. RadixSort is stable and does not sort in-place.*

Proof. RadixSort uses d rounds of CountSort by viewing each key as a d -digit radix- k integer and sorts the keys by sorting the keys in rounds, each round i involving sorting using CountSort and with key value the i -th rightmost digit.

```
RadixSort(I, 0, n, k, d)
1. for(i=0 ; i < d ; i++ ) {
2.   CountSort(I, 0, n, k) on i-th rightmost digit;
3.   I=0;
4. }
```

The algorithm has d iterations each one requiring $O(n + k)$ steps (line 2) i.e. a total $O(d(n + k))$ (for lines 1-2). All properties of CountSort are inherited to/by RadixSort. One may think that starting the sorting from the most significant position is the way to go. Then we are going to perform d iterations of the following loop: starting from the 1-st most significant position and sorting the keys according to that position's digit value using count sort and generating k subsequences catenated one after the other in the output. Total time is $O(n + k)$. We then sort with the 2-nd 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 $O(k^2)$ (a set of k counters for each subsequence) and the scanning of input keys an additional $O(n)$ for a total $O(k^2 + n)$. When the least significant digit is reached, the time to sort the n keys according to that digit will be $O(n + k^d)$. The reason for this failure of RadixSort is that it starts from the most significant digit. In the pseudocode we do not use the $O(n + k^d)$ approach: we start from the least significant digit and past ordering is ignored in later phases to maintain a $\Theta(d(n + k))$ running time.

Distribution-based sorting

Array-based RadixSort

Proof. We thus perform radix sort instead using d phases and starting with the least significant digit 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 based on digits $l..1$. But for this to hold, it is essential that the CountSort operation is stable, and it is so! For example, if three inputs are 21, 22, 23 have already been sorted on their rightmost digit (and thus 21 appears before 22 and then before 23 because $1 < 2 < 3$) we want the relative order for those keys to be maintained i.e. 21 appears before 22 before 23 when we sort them on the most significant digits that is 2 for all three of them. A non-stable algorithm could output 23, 21, 22 when sorting on the leftmost digit. A stable sorting algorithm with the output 21, 22, 23 maintains all previous orderings (of the same valued digit currently considered). ■

Example 12. An example of RadixSort is shown next.

Proof.

Input is : 358, 137, 246 and arranged vertically
Output sorted means: top : smallest , bottom : largest
Leftmost bit To Rightmost bit Operation (DOES NOT SORT)
3 5 8 1 3 7 1 3 7 2 4 6 notmin used as key for countsort
1 3 7 --> 2 4 6 --> 2 4 6 --> 1 3 7 min
2 4 6 3 5 8 3 5 8 3 5 8 max
Rightmost bit To Leftmost bit Operation (IT SORTS)
 V V V
3 5 8 2 4 6 1 3 7 1 3 7 min
1 3 7 --> 1 3 7 --> 2 4 6 --> 2 4 6
2 4 6 3 5 8 3 5 8 3 5 8 max

Array-based Radixsort Example

Example 13. An example of wrong-direction RadixSort is shown next.

Proof.

Input is : 152, 430, 151 and arranged vertically
Output sorted means: top : smallest , bottom : largest
Leftmost bit To Rightmost bit Operation (DOES NOT SORT)
1 5 2 1 5 2 4 3 0 4 3 0 notmin
4 3 0 --> 1 5 1 --> 1 5 1 --> 1 5 1 min
1 5 1 4 3 0 1 5 2 1 5 2 notmax

■

DRAFT

Array-based Radixsort

Example

Fact 1. The integer numbers $0, \dots, m - 1$ can be represented by $\lceil \lg m \rceil$ bits.

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

Fact 3. Therefore, with $\lg y$ bits we can represent $2^{\lg y} = y$ values $0, \dots, y - 1$.

Exercise 4. What is 110?

Proof.

- 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, decimal) 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 $0, \dots, k - 1$ in time $\Theta(n + k)$.

What is linear time sorting?
CountSort vs RadixSort vs ?

Fact 5. CountSort sorts n keys in the range $0, \dots, k^d - 1$ in time $O(n + k^d)$.

Fact 6. RadixSort sorts n keys in the range $0, \dots, k^d - 1$ in time $O(d(n + k))$.

Fact 7. Suppose we have n keys that 32-bit (unsigned) integers. RadixSort requires:

- 1-digit keys: View each key as an 1-digit radix-2³² integer i.e. $k = 2^{32}$, $d = 1$. Running time is $\Theta(1 \cdot (n + 2^{32}))$.
- 2-digit keys: View each key as an 2-digit radix-2¹⁶ integer i.e. $k = 2^{16}$, $d = 2$. Running time is $\Theta(2 \cdot (n + 2^{16}))$.
- 4-digit keys: View each key as an 4-digit radix-2⁸ integer i.e. $k = 2^8$, $d = 4$. Running time is $\Theta(4 \cdot (n + 2^8))$.
- 32-digit keys: View each key as an 32-digit radix-2 integer i.e. $k = 2$, $d = 32$. Running time is $\Theta(32 \cdot (n + 2^1))$.

What is the best way to sort 32-bit unsigned integers? Say 1,000,000 of them?

Proof. The first variant of radix-sort with $d = 1$ is in fact CountSort. We observe that all variants are $\Theta(n)$ but the constants hidden are too large sometime or too small sometimes. Consider HeapSort and MergeSort instead. They are both $\Theta(n \lg n)$ in term of running time.

Yet for $n = 1000000$ things are different. One cannot compare numbers related to countsort operations to numbers related to comparison counts. Only implementations counts. Asymptotics go out of the window for fixed values of n such as $n = 1000000$ unless we have done a thorough accounting before hand. ■

What is linear time sorting?

CountSort vs RadixSort vs ?

Exercise 5. We have n keys in the range $0, \dots, n^2 - 1$. How do we sort them?

Proof. If we use MergeSort/HeapSort 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 RadixSort helps. Each integer in the range $0, \dots, n^2 - 1$ 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 RadixSort running time is $O(d(n + k)) = O(2(n + n)) = O(n)$. RadixSort beats MergeSort/HeapSort and CountSort. ■

Exercise 6. We have n keys in the range $0, \dots, 2^n - 1$. How do we sort them?

Proof. If we use MergeSort/HeapSort 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 RadixSort helps. Each integer in the range $0, \dots, 2^n - 1$ 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 RadixSort running time is $O(\frac{n}{\lg n}(n + k)) = O(\frac{n}{\lg n}(n + n)) = O(n^2/\lg n)$. RadixSort/CountSort are beaten by MergeSort/HeapSort, though they still beat in the worst case InsertionSort/QuickSort etc. ■

Problem 4. We throw n balls independently and randomly of each other 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 $E[n_i^2]$ (the expected value of n_i^2).

Solution. We introduce some notation first. Let $n_i, 1 \leq i \leq n$ be the number of balls falling into the i -th bin. Let $Y_{ij}, 1 \leq i, j \leq n$ be an indicator 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, $Y_{ij} = 0$. Then $n_i = \sum_j Y_{ij}$. Ball j falls into bin i with probability $1/n$. Therefore $\text{Prob}(Y_{ij} = 1) = 1/n$ and thus $\text{Prob}(Y_{ij} = 0) = 1 - 1/n$. Moreover

$$E[Y_{ij}] = 1 \cdot \text{Prob}(Y_{ij} = 1) + 0 \cdot \text{Prob}(Y_{ij} = 0) = 1 \cdot (1/n) + 0 \cdot (1 - 1/n) = 1/n.$$

In addition

$$E[Y_{ij}^2] = 1^2 \cdot \text{Prob}(Y_{ij} = 1) + 0^2 \cdot \text{Prob}(Y_{ij} = 0) = 1 \cdot (1/n) + 0 \cdot (1 - 1/n) = 1/n.$$

And as a result we also have by the definition of variance the following.

$$\text{var}[Y_{ij}] = E[Y_{ij}^2] - E^2[Y_{ij}] = 1/n - 1/n^2 = (1/n)(1 - 1/n).$$

By way of definition

$$E[n_i] = E\left[\sum_j Y_{ij}\right] = \sum_{j=1}^n E[Y_{ij}] = n(1/n) = 1.$$

and the fact that the expectation of the sum is the sum of the expectations, we have

$$\text{var}[n_i] = \text{var}\left[\sum_j Y_{ij}\right] = \sum_j \text{var}[Y_{ij}] = n(1/n)(1 - 1/n) = 1 - 1/n.$$

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 $k, k \neq j$.

$$\text{var}(n_i) = E[n_i^2] - E^2[n_i].$$

Therefore we conclude that

$$E[n_i^2] = \text{var}(n_i) + E^2[n_i] = (1 - 1/n) + 1 = 2 - 1/n.$$

■

Probabilistic sorting

A BucketSort that is not our BucketSort

Theorem 15 (ProbabilisticSort a.k.a BucketSort). *Sorting n real-values keys that are uniformly at random drawn from the range $[a, b]$ (or equivalently the range $[0, 1]$) requires expected linear time $\Theta(n)$. The running time is though $T(n) = O(n^2)$. The algorithm is sometimes known as ProbabilisticSort or BucketSort.*

Mapping from $[a, b]$ into $[0, 1]$. For an $x \in [a, b]$ we map it to $y = (x - a)/(b - a) \in [0, 1]$.

Note the same name BucketSort refers to two different algorithms that solve two different sorting problems. Read the definition of the problem rather than the name of the algorithm. The idea is to divide the interval into n equally spaced subintervals that will be called **buckets**. Let us index buckets $0, \dots, n - 1$ left to right. We then assign each key to its destination bucket (may be we have a linked list to collect all keys that are destined for the same bucket). We sort the keys per bucket (linked list to array) using our favorite comparison and exchange based algorithm. For two buckets $i < j$ the elements of bucket i are smaller than those of j . Therefore after the comparison sort all keys are sorted and the output is generated by concatenating the buckets. The algorithm is shown below. Input is $A[0..n - 1]$ and buckets are represented by linked lists with heads in array $L[0..n - 1]$.

```
ProbabilisticSort(A,n)
0. Initialize n Linked lists (Buckets) L[i]; // L[i] [i/n..i+1/n]
1. for(i=0;i<n;i++)
2.   A[i] belongs to bucket L[floor(n A[i])]: insert it into L
3. for(i=0;i<n;i++)
4.   sort L[i] with InsertionSort
5. A= Concatenate L[0] ... L[n-1];
6. release the space of L[i];
7. return(A);
```

So named BucketSort Example

Example 14. An example of ProbabilisticSort is shown next.

Proof.

Input is : 0.48, 0.31, 0.25, 0.13, 0.92 i.e. $n=5$

Input 0.48 --goes to bin --> $\text{floor}(0.48 \cdot 5) = 2$

Input 0.31 --goes to bin --> $\text{floor}(0.31 \cdot 5) = 1$

Input 0.25 --goes to bin --> $\text{floor}(0.25 \cdot 5) = 1$

Input 0.13 --goes to bin --> $\text{floor}(0.13 \cdot 5) = 0$

Input 0.92 --goes to bin --> $\text{floor}(0.92 \cdot 5) = 4$

```

0      0.2    0.4    0.6    0.8    1
|-----|-----|-----|-----|-----|

```

Bin 0	Bin 1	Bin 2	Bin 3	Bin 4	A bin ... is a bucket! (name fits space!)
-------	-------	-------	-------	-------	--

0.13	0.31 0.25	0.48	-	0.92
------	--------------	------	---	------

```

| |
| |

```

Sort	V	Smallest	Top of bucket
buckets		Largest	Bottom of bucket

0.13	0.25 0.31	0.48	-	0.92
------	--------------	------	---	------

```

| |
| |

```

Concatenate	V	Concatenate(0,1,2,3,4)
buckets		

Output is: 0.13, 0.25, 0.31, 0.48, 0.92 i.e. $n=5$



So named BucketSort

Expected running time

Proof. 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 (i.e. in list $L[i]$). Each key as it is drawn uniformly at random has probability $p_i = 1/n$ to fall into $L[i]$. 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

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

Then, by taking expectations on both sides we get

$$\begin{aligned} E[T(n)] &= \Theta(n) + E\left[\sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + E\left[O\left(\sum_{i=0}^{n-1} n_i^2\right)\right] \\ &= \Theta(n) + O\left(\sum_{i=0}^{n-1} E[n_i^2]\right) \\ &= \Theta(n) + O\left(\sum_{i=0}^{n-1} (2 - 1/n)\right) \\ &= \Theta(n) + O(n(2 - 1/n)) \\ &= \Theta(n) + O(2n - 1) \\ &= \Theta(n). \end{aligned}$$

■

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem 1.

TRUE or FALSE?

- QuickSort is stable.
- RadixSort is stable.
- CountSort is stable.
- CountSort sorts in-place.
- We can optimally in-place sort n keys in the range $0, \dots, n-1$ by using count-sort (i.e. one round of radix-sort).
- Among count-sort, insertion-sort, quick-sort, and merge-sort the most efficient way to in-place sort n keys in the range $0, \dots, n-1$ is insertion-sort.

Proof. FALSE, TRUE, TRUE, FALSE, TRUE, TRUE.

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 2.

Name the most time-efficient algorithm and specify its worst-case running time among **HeapSort**, **CountSort** and/or **RadixSort** that solves each one of the problems below. If **RadixSort** is the answer, mention how many iterations of **CountSort** are involved in it. In case of a tie, a stable beats a non-stable algorithm. Indicate running time as needed.

1. Sorting n keys in the range $0, \dots, n - 1$.
2. Sorting n keys in the range $0, \dots, 2^n - 1$.
3. Sorting n keys in the range $0, \dots, n^{\lg n} - 1$.
4. Sorting n keys in the range $0, \dots, n^2 - 1$.

Proof. (1) CountSort $\Theta(n)$, (2) HeapSort $O(n \lg n)$, (3) RadixSort, $\lg n$ rounds, $O(n \lg n)$ beats in a tie-breaker HeapSort, (4) RadixSort, 2 rounds, $\Theta(n)$. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's webpage

Problem 3.

How could you modify quicksort (give pseudocode or describe convincingly the changes) so that its worst-case performance is $O(n \lg n)$? Describe your changes, analyze the worst-case running time of the modified quicksort and show that it is indeed $O(n \lg n)$ in the worst-case. (You need Subject 7 material. Skip it until we cover Subject 7! Also available as Problem 5 there! It is more suited as a review problem at the end of the semester!)

Proof. Pick the splitter (pivot) key to be the median of the keys of the input participating in the recursive call. Finding the median requires linear time (though with a very large constant). However it guarantees a 50-50 split (plus or minus one key). Thus the recurrence for the running time of this modified quick-sort will be (plus or minus one key): $T(n) = 2T(n/2) + Cn$, where C is a big constant. The asymptotic solution is $\Theta(n \lg n)$, which is acceptable and asymptotically faster than $\Theta(n^2)$ in the worst-case. Pseudocode-wise we can just change line 1 of Partition slightly.

OLD CODE

```
1. splitter= A[r];
```

NEW CODE

```
// DSelect 'returns' the index of the statistic not the statistic itself
1a. splitter-index=DSelect(A,floor((r-l+1)/2),l,r);
1b. swap(A[r],A[splitter-index]);
1c. splitter=A[r];
```

■

Problem Set
Problem 4.

Answer the following questions briefly by justifying your answers. If needed provide asymptotically tight bounds as answers. ('BucketSort' is the probabilistic version of BucketSort; not the linked list version of CountSort).

- (a) What is the worst-case running time of 'BucketSort' ?
- (b) What is the running time of 'BucketSort' ?
- (c) What is the expected running time of 'BucketSort' ?

Proof. (a) $T(n) = \Theta(n^2)$.

(b) $T(n) = O(n^2)$.

(c) $T(n) = \Theta(n)$.

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 5.

Answer the following questions for the described variants of quicksort and JUSTIFY your answers.

- (a) Suppose Partition picks the splitter to be the key from the last position of the subarray. On a sorted array does quicksort now take $\Theta(n)$, $\Theta(n \lg n)$ or $\Theta(n^2)$? Explain.
- (b) Suppose Partition picks the splitter to be the key from the middle position of the subarray. On a sorted array does quicksort now take $\Theta(n)$, $\Theta(n \lg n)$ or $\Theta(n^2)$? Explain.
- (c) Suppose Partition picks the splitter to be the median of the first three keys of the subarray. [The median of three keys is the middle value; i.e. the median of 6, 1, 11 is 6.] On a sorted array does quicksort now take $\Theta(n)$, $\Theta(n \lg n)$ or $\Theta(n^2)$? Explain.
- (d) Suppose Partition picks the splitter to be the media of the first, last and middle keys of the subarray. On a sorted array does quicksort now take $\Theta(n)$, $\Theta(n \lg n)$ or $\Theta(n^2)$? Explain.

Proof. (a) We are talking about GT so $T(n) = \Theta(n^2)$. Check Subject 6 for a proof.

(b) We are talking about GT so $T(n) = \Theta(n \lg n)$. Check Subject 6 for a proof of best case running time which is the case with such a splitter and input.

(c) We are talking about GT so $T(n) = \Theta(n^2)$. Not much difference than (a)

(d) You do it!

■

Problem Set
Problem 6.

Give an efficient algorithm that works in-place which reorders an array A of n integer keys so that all negative keys appear before the positive keys in the output sequence. Justify your answer. Your algorithm must be **efficient** (i.e. one can not give an algorithm whose asymptotic worst-case performance can be improved) and **should work in place** (i.e. you can not use an extra array, and in summary the extra space you use besides the one needed for storing A is constant).

Proof. Think! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 7.

Professor I. M. Good devised a variant of Partition, called GoodPartition that guarantess a 10%-90% split around the splitter no matter what the input sequence is. The new version of quicksort that uses Good Partition is shown below.

```
GoodQuickSort(A,n)
1. GoodQuicksort(A,0,n-1);

GoodQuicksort(A,l,r);
1. if (l<r) {
2.  m=GoodPartition(A,l,r); // splitter has become A[m]
3.  GoodQuicksort(A,l,m-1); // sort recursively all keys <= A[m]
4.  GoodQuicksort(A,m+1,r); // sort recursively all keys >= A[m]
}
```

What is the running time of GoodQuicksort on any input sequence of n keys? Justify your answer.

Proof. Then the running time of GoodQuickSort would be $T(n)$ where $T(n) = T(n/10) + T(9n/10) + \Theta(n)$. Using the substitution method, and the proofs of PS2, we obtain $T(n) = \Theta(n \lg n)$. ■

Problem Set

Problem 8.

We have 4 coins all of the same weight except one that is fake and may weigh LESS or MORE than the other coins (we don't know what the case is). We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (or less). If two coins are weighed (one on each side of the scale) and they do not weigh the same, we can't decide by this weighing alone which of the two is the fake.

Can you find the fake coin with only 2 weighings? Explain (and justify your answer).

Proof. We indicate a weight of A and B by showing A:B. As stated, the problem has 4 states (i, F) , where i is the coin label and F its status as fake. (Implied is that the other coins are genuine.) With two weighing (Equal or NotEqual) we are done. The decision tree is binary and of height logarithmic to the leaves/states. Here is how to find the fake coin.

A:B --- == ---> (C or D is F)--- A:C --- == D is F; A,B,C genuine
 |
 |
 --- != ---> (A or B is F)--- A:C --- == B is F; A,C,D genuine
 |
 --- != ---> A is F; B,C,D genuine

We now rephrase the problem by also asking to find whether the fake coin is lighter or not and allowing for the possibility that all are genuine. The problem has 9 states: eight of the form (i, S) where i is the coin label of a fake coin and S an L for lighter or H for heavier with the ninth state being all-g i.e. all are genuine. As $\log_3(9)$ is still a two we can use a ternary decision tree and try with two weighings with it. A lower bound does not mean it is an upper bound as well. Try three if you can't do it with two weighings! ■

Problem Set
Problem 9.

We have 27 coins all of the same weight except one that is fake and weighs LESS than the other coins. We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (or less). Can you find the fake coin with only 3 weighings? Explain (and justify your answer).

Proof. Split into 3 groups. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 10.

- a. A new version of quicksort has been suggested by Professor I. M. CRAZY called CRAZYSORT in which an adversary picks the splitter in Partition in the worst possible way to maximize the running time of CRAZYSORT, that is, line 1 of Partition (see textbook) is determined by the adversary in constant time $O(1)$. What is the running time of CRAZYSORT? Justify your answer.
- b. A new version of quicksort is proposed by Professor I. M. PARANOID called PARANOIDSORT in which an adversary picks the best, instead of the worst, possible splitter in line 1 of Partition *every other time* (i.e. he alternates between best and worst splitter). What is the running time of PARANOIDSORT? Justify your answer.
- c. The bounds for the running time in parts (a) and (b) MUST be asymptotically tight.

Proof. Flip coins maybe! I am kidding... Think about randomization bad-good-bad-good! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 11.

How to turn any sorting algorithm that is not stable into a stable sorting algorithm by paying a space penalty and also a comparison penalty.

Proof. If input sequence consists of keys $x_1, \dots, x_n$ extend each key by a pair (x_i, i) , where second element gives position in input sequence. This makes all keys distinct. Sort the extended keys (pairs) where a key is a pair, not just x_i . This scheme may at most double the running time and uses $O(n)$ additional memory. Two keys (x_i, i) and (x_j, j) are compared as follows. If $x_i \neq x_j$ report the result of the comparison of x_i and x_j ; if $x_i = x_j$ report the result of the comparison to be that of i and j . We increase the input by n integer index values or n integer pointer values. The comparison count also potentially doubles; not however when we compared indexes we do a subtraction not call the compare function a second time! Suppose the input sequence $x_1, \dots, x_n$ is to be sorted in non-decreasing order. As some x_i 's may be equal we create for key x_i a new key which is the pair (x_i, i) . The first element of the pair is x_i and the latter is the index/position of x_i in the input sequence. The input to the sorting algorithm is going to be not $x_1, \dots, x_n$ but $(x_1, 1), \dots, (x_n, n)$. Note that the keys of the newly formed input are always distinct; even if $x_i = x_j$ for some $i \neq j$, the second elements of the two pairs are distinct as $i \neq j$. As any of the sorting algorithms works fine with distinct keys (ie stability is no issue then), it will sort the n augmented keys. If key x_i was before x_j in the input sequence, it means $i < j$ and therefore $(x_i, i) < (x_j, j)$, and therefore in the output sequence (x_i, i) will appear before (x_j, j) . When the output sequence is obtained, we reobtain the original keys from the pairs (by reading off the first element of the pair) to get the output sequence for the original problem. ■

Problem Set
Problem 12.

How to turn QuickSort into an $O(n \lg n)$ running time algorithm.

Proof. In QuickSort make splitter to be the median of the input sequence. This way step 1 of GT Partition, $splitter = A[r]$ becomes $splitter = Median(A, l, r)$ and then $swap(A[splitter], A[r])$ similarly to randomized quicksort. (Median returns the index not the value of the median in A). Median selection takes $O(n)$ time i.e. asymptotically the same time as the remainder of Partition. Given that $splitter$ is the median the two sets split by x are of about the same size $(r - l + 1)/2$, and the recurrence for the new quicksort is that of merge sort i.e. running time becomes $O(n \lg n)$ in the worst case. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 13.

In an earlier problem set you were asked to sort an array of 0 and 1. The following solution was proposed.

```
Sort(Y[1..n],n)
0. count=0;
1. for (i=1;i<=n;i++)
2.   if (Y[i]== 0) count++;
3. for (i=1;i<=count;i++)
4.   Y[i]=0;
5. for (i=count+1;i<=n;i++)
6.   Y[i]=1;
```

Now array Y contains not only $Y[i].k$ the key value that is 0 or 1, but also contains $Y[i].info$. Think about k being an indicator of PhD student or MS student, and $Y[i].info$ being the last name of a student (no students of the same matriculation level have the same last name!). Give a time and space efficient algorithm that sorts by last name but PhD students appear last and MS students first!

Proof. The count works up to a point. Output needs to be filled as in CountSort. ■

Problem Set
Problem 14.

Is it possible to print the n keys of a MaxHeap in sorted order by performing $\Theta(n)$ comparisons? Explain and Justify your answer.

Proof. Suppose that it was possible to print the keys of a heap in sorted order in $O(n)$ worst-case time. As MakeHeap requires $O(n)$ worst case time, and printing a heap $O(n)$ more, sorting n keys would thus take $O(n)$ time in the worst case, a contradiction to the comparison-tree derived lower bound of $\Omega(n \lg n)$. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 15.

Expected number of assignments in line 4 of MIN or (in the example below) MAX. (Assume keys are distinct and 'random').

```
MAX(A[1..n])
1. M = A[1];
2. for (i=2; i <= n; i++)
3.   if A[i] > M
4.     M = A[i];
5. return(M);
```

Proof. Among the n distinct numbers only one number can be the maximum. The probability that M randomly chosen is this maximum is $1/n$.

When line 4 is executed M is set to $A[i]$, i.e. $A[i]$ is the maximum of all keys $A[j]$, where $1 \leq j \leq i$, and therefore $A[i] \geq A[j]$. Fix a value for i . The probability that line 4 is executed is the probability that $A[i]$ is the maximum of all i keys $A[j]$, $j \leq i$. This is $1/i$: the probability that $A[i]$ is the max of $A[1] \dots A[i]$.

Line 4 is either executed or not, i.e. $M_i = 1$ or $M_i = 0$. The probability that $M_i = 1$ is given by $1/i$. The probability that $M_i = 0$ is 1 minus the probability that $M_i = 1$.

$$E[M_i] = 1 \cdot \text{Prob}(M_i = 1) + 0 \cdot \text{Prob}(M_i = 0) = 1 \cdot (1/i) + 0 = 1/i.$$

Let $M = M_1 + M_2 + \dots + M_n$. M indicates the number of times line 4 was executed. Then by the linearity of expectation we have that

$$E[M] = E[M_1 + M_2 + \dots + M_n] = E[M_1] + E[M_2] + \dots + E[M_n] = 1 + 1/2 + 1/3 + \dots + 1/n = \ln n + O(1)$$

where the last term is by Euler's gamma function. ■

Problem Set
Problem 16.

Which algorithm among heap-sort, radix-sort, count-sort is asymptotically faster for each one of the following sorting problems. Justify your answer. In case of a tie, the tie-breaker is an in-place algorithm.

- (a) Sorting of n keys, where each key takes values in the range $0 \dots n^4 - 1$.
- (b) Sorting of n keys, where each key takes values in the range $0 \dots n^{2 \lg n} - 1$.
- (c) Sorting of n keys, where each key takes values in the range $0 \dots n^n - 1$.

Proof. (a) We use 4-digit radix- n sorting of the keys in the range from 0 to $n^4 - 1$. We use 4 rounds of count sort (i.e. radix-sort) for a total time of $O(4n) = O(n)$. RadixSort beats CountSort that would require $O(n^4)$ or heap-sort.

(b) If we use $2 \lg n$ -digit radix- n sorting of the keys we need time $\Theta(\lg n)$ rounds of count-sort for sorting keys in the radix- n range i.e. $O(n)$ time per round for the RadixSort algorithm. Total is $\Theta(n \lg n)$. (One-round) CountSort requires $O(n^{2 \lg n})$ i.e. too much. HeapSort on the other hand also takes $O(n \lg n)$.

There is a tie between HeapSort and RadixSort. The tie-breaker is the in-place property of HeapSort. HeapSort wins.

(c) If we use RadixSort for the keys in the range from 0 to $n^n - 1$, we need n rounds of CountSort i.e. $\Theta(n^2)$ time since we will view the keys as n -digit radix- n integers. CountSort itself requires too much time for initialization alone $\Theta(n^n)$. HeapSort however uses only $O(n \lg n)$. The winner is HeapSort by a margin. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 17.

Suppose that the splits at every level of quicksort are in the proportion 0.9 to 0.1, i.e. at every level 90 percent of the keys go to the left set and 10 percent to the right. Show that the minimum depth of a leaf in the recursion tree of quicksort is approximately $-\lg n / \lg 0.1$ and the maximum depth is approximately $-\lg n / \lg (0.9)$. Don't worry about integer round-off.

Proof. Since we have a 10-percent to 90-percent split, this means that the recurrence for the running time becomes

$$T(n) = T(0.1n) + T(0.9n) + \Theta(n)$$

A solution based on PS2 Problem 19-20 is $T(n) = \Theta(n \lg n)$. To summarize using $a = 0.1$ below, for the recursion tree based solution, the leftmost path of the recursion tree involves nodes with problem size $n, an, a^2n, a^3n, \dots$ and the rightmost path $n, (1-a)n, (1-a)^2n, (1-a)^3n, \dots$

How long is the leftmost path? It is L long if $a^L n = 1$. Taking logarithms base two $L \lg a + \lg n = 0$, i.e. $L = -\lg n / \lg a$. Substitute 0.1 for a and we get the desired answer.

How long is the rightmost path? It is R long if $(1-a)^R n = 1$. Taking logarithms base two $R \lg (1-a) + \lg n = 0$, i.e. $R = -\lg n / \lg (1-a)$. Substitute 0.1 for a or 0.9 for $1-a$ and we get the desired answer. If the depth of the recursion tree is $O(\lg n)$ and at every node the time is $\Theta(n)$, the result follows. ■

Problem Set
Problem 18.

Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance? Explain.

Proof. There is no reason to analyze its worst-case performance. Randomized quick-sort for example has the same worst-case performance as ordinary quick-sort and randomization does not change this. If the choices of the random number generator lead to the worst-case, that's it, we get worst-case performance. The introduction of randomness, however, makes this (possibility of observing worst-case performance) **highly unlikely**. Much more interesting is an average-case analysis that analyzes the performance that "most likely" will be observed. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 19.

If we have some prior information about the ordering of the keys of the input, i.e. if they are "almost sorted" we could readjust quicksort to take this into consideration. When quicksort operates on a subsequence with fewer than m keys, let it just return without sorting the subsequence. As soon as the quicksort returns from its top-level call (the one with arguments 0 and $n - 1$), invoke insertion sort on the entire "output" sequence. Argue that this two-stage sorting algorithm i.e. the modified quick-sort followed by insertion-sort runs in expected time $O(nm + n \lg(n/m))$. Find an m that minimizes this expected time.

Proof. We can change the high-level quicksort algorithm into the new one below as follows; execute in step 1 the modified quick sort that stops at m keys (called QUICKSORT2) and then run InsertionSort on the output of QUICKSORT2 (all keys). At recursion depth 1, the original input is split into two halves of expected size $n/2$. At recursion depth i , the input is split into 2^i sets each one having an expected $n/2^i$ keys. If $i = \lg(n/m)$ then we have n/m sets ($2^i = n/m$ for $i = \lg(n/m)$) with expected number of keys per set of $n/2^i = n/(n/m) = m$. The sequences of these keys are left unsorted and dealt with by Insertion Sort. What is the expected running time of step 1 of QUICKSORT? Each level of QUICKSORT takes $O(n)$ steps and there are $\lg(n/m)$ recursive calls for a total of $O(n \lg(n/m))$ expected running time for step 1 of QUICKSORT. InsertionSort takes over and sorts the keys of the n/m sets. Let us call the subsets of A $s_1, s_2, \dots, s_{n/m}$. Because of the way quicksort works, all keys in s_1 are less than all keys in s_2 and all keys in s_2 are less than all keys in s_3 and so on. So every key is at most m positions away from its destination within its own set. Therefore insertion sort will spend time $O(m)$ per key (a key in set s_i can only move at most m positions in that set since the set is of size m), and since there are $O(n)$ keys the total time of Insertion sort (step 2 of QUICKSORT) will only be $O(nm)$ not $O(n^2)$. Combining the contributions of steps 1 and 2 of QUICKSORT we get that the expected running time is $O(nm + n \lg(n/m))$. Due to the first term we want m to be up to $\lg n$ so that the first term becomes $O(n \lg n)$. The second term is for this range of m also $O(n \lg n)$. Values of $m = \omega(\lg n)$ make the expected running time of QUICKSORT (modified version) $\omega(n \lg n)$; these values are disregarded. In practice the best value of m is chosen by trial and error trying values from $k = 1$ all the way to $\lg n$ or some $A \lg n$ for some constant $A > 1$, and the best one is used after experimentation.

Problem Set Continued

```
QUICKSORT(A,n)
1. QUICKSORT2(A,0,n-1);
2. InsertionSort(A,n);

QUICKSORT2 (A,l,r)
1.  if (r-l+1) >= m          //A[l..r] has r-l+1 keys ; do nothing if r-l+1 < m.
2.    then  m = PARTITION(A,l,r)          otherwise sort as before.
3.        QUICKSORT2(A,l,m-1)
4.        QUICKSORT2(A,m+1,r)
```

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 20.

You are given a set A of n numbers (real, integer, positive, or negative) out of which only m are distinct where $m \leq \sqrt{n}$. Find the m distinct values and print them in sorted order. For example if A looks like

$\{\pi, e, 5, 10^7, \pi, 10^7, e, 5, 10^7, e, \pi, 5, 10^7, e, e, 10^7\}$

we have $n = 16$ but $m = 4$. Your algorithm should be faster than an $O(n \lg n)$ generic algorithm for a variety of choices of m . For example, if $m = O(1)$ your algorithms should be $O(n)$ which is faster than $O(n \lg n)$. Similarly if $m = O(\lg n)$.

Additional question. How would you sort the n keys in the same asymptotic time?

Hint. Give an $O(n \lg m)$ algorithm.

Proof. No solution is provided. Great as an interview question though. There are several ways to deliberate about this problem.

(a) Use arrays and ... Countsort-type methods. But beware: CountSort, BucketSort or RadixSort can not be used. Keys might not be integers. They might be arbitrary key values. Moreover no one says that the keys are from a given range of k or m consecutive values $0 \dots m - 1$. If this is a distraction

(b) a solution that is not space efficient or space optimal is available with a follow-up Problem Set. However the problem is completely rephrased! So you have got to figure out what problem it is!

■

Problem 21.

RadixSort with HeapSort? RadixSort of n keys in the range $0 \dots k^d - 1$ done in class involved d rounds of CountSort from least significant to most significant digit in time $\Theta(d(n + k))$. We decide instead of using CountSort to use an in-place sorting algorithm and in particular HeapSort.

- (a) What would the running time of RadixSort be if HeapSort replaces CountSort. Justify your answer.
- (b) Would RadixSort (with HeapSort in place of CountSort) work correctly then? That is would it sort arbitrary integers keys in the range $0 \dots k^d - 1$? Explain.

Proof. For (a) you are allowed to change the structure of the heap to use the CountSort available information. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 22.

Facepeek. You are an early adopter (or is it adoptee) of a social network called Facepuke. You have lots of friends, in fact n of them; each friend is represented by a 64-bit integer. You want to sort ... out ... your friends. What sorting algorithm would you use and what would its running time be? (Give name and running time only, using as tight asymptotic notation as possible.)

Proof. Radix-sort, $\Theta(n)$, around 8 or 4 rounds. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 23.

LunaticSort. A new version of quicksort is proposed by Professor I. M. NJIT, called LunaticSort in which an adversary picks the best, instead of the worst, possible splitter in line 1 of Partition *every 5-th time*, that is four times in a row a worst possible splitter is picked but then the fifth time the best possible splitter is picked. What is the running time of LunaticSort? Justify your answer.

Proof. LunaticSort? Didn't we encounter CrazySort or ParanoidSort before?

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem 24.

Book-keeping. The NJIT library has N books and every book belongs to one of M publishing houses. Array $A[1..N]$ maintains the publisher of a book, i.e. if $A[i] = j$, then the publisher of book i is j , where $j = 1, \dots, M$. Give an algorithm whose worst-case running time is $O(N + M)$ that calculates the number of books belonging to publisher j .

Proof. CountSort type. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page