Let $t(n)$ denote the average number of (array) comparisons by RandSelect$(A, 1, n, i)$

Assume that each permutation of $A[1..n]$ is equally likely, Hence the return value of RandPartition is equally likely to be any of the numbers $1 \le q \le n$.

A priori $t(n)$ depends on i. We'll see that in fact it doesn't. Recall that RandPartition does $(n-1)$ comparisons. Thus

$$t(n) = \frac{\sum_{q=1}^{n} \left((n-1) + P(i<q)\, t(q-1) + P(i>q)\, t(n-q) \right)}{n}$$

where

$$P(i<q) = \frac{n-i}{n} \quad \text{and} \quad P(i>q) = \frac{i-1}{n}$$

are the probabilities that q is in the range $i < q \le n$ and $1 \le q < i$ respectively. Thus

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

$$= (n-1) + \frac{1}{n^2} \left[(n-i) \sum_{q=1}^{n-1} t(q) + (i-1) \sum_{q=1}^{n-1} t(q) \right]$$

$$= (n-1) + \frac{1}{n^2}(n-i+i-1)\sum_{q=1}^{n-1} t(q)$$

$$\therefore \boxed{t(n) = (n-1) + \left(\frac{n-1}{n^2}\right)\sum_{q=1}^{n-1} t(q)}$$

THEREFORE $t(n)$ DOES NOT DEPEND ON i, AS CLAIMED EARLIER.

OBSERVE THAT THIS RECURRENCE IS VERY SIMILAR TO THE ONE FOR THE AVERAGE RUN TIME OF Quicksort.

EXERCISE

SHOW THAT $t(n) = \Theta(n)$.

OBVIOUSLY $t(n) \geq n-1 = \Omega(n)$. PROVE THAT $t(n) = O(n)$ BY INDUCTION ON n. i.e. SHOW

$$\forall n \geq 1 \; : \; t(n) \leq 2n$$

INDUCTION HYPOTHESIS : $\forall\, q \leq n-1 : t(q) \leq 2q$

$$\therefore t(n) \leq (n-1) + \left(\frac{n-1}{n^2}\right)\sum_{q=1}^{n-1} 2q$$

$$= (n-1) + \left(\frac{n-1}{n^2}\right)\cdot 2\,\frac{n(n-1)}{2} \leq 2n$$

↑ PROVE.

EXERCISE

FIND THE EXACT SOLUTION TO THIS RECURRENCE.

ANSWER:

$$t(n) = (n-1)\left\{1 + \sum_{r=1}^{n-1} \frac{\mu_r \cdot (r-1)}{\mu_n \cdot n(n^2+n-1)}\right\}$$

WHERE

$$\mu_n = \prod_{K=2}^{n} \left(\frac{K}{K^2+K+1}\right)$$

OTHER SORTING ALGORITHMS (8.1-8.4)

THE SORTING ALGORITHMS WE'VE STUDIED SO FAR USE NO INFORMATION OTHER THAN THE (TOTAL) ORDER RELATION ON $A_1, \ldots, A_n$.

I.E. THEY HAS THE ABILITY TO COMPARE TWO ARRAY ELEMENTS $A_i < A_j$, AND OBTAIN TRUE OR FALSE. NO OTHER INFORMATION ABOUT THE RELATIONSHIP BETWEEN ARRAY ELEMENTS WAS USED.

SUCH AN ALGORITHM IS CALLED A COMPARISON SORT.

NON-COMPARISON SORTS ARE THOSE WHICH HAVE EXTRA INFORMATION AT THEIR DISPOSAL.

THEOREM.

ANY COMPARISON SORT PERFORMS $\Omega(n \lg n)$ COMPARISONS ON INPUT OF length n, IN WORST CASE.

PROOF LATER.

COMPARISON SORTS:

- BubbleSort
- InsertionSort
- HeapSort
- MergeSort
- QuickSort

NON-COMPARISON SORTS:

- Counting Sort
- RadixSort
- BucketSort

Counting Sort

THIS ALGORITHM ASSUMES THAT THE ELEMENTS OF A ARE <u>INTEGERS</u> IN THE RANGE 0 TO K FOR SOME $K \in \mathbb{Z}_+$

$$0 \leq A[1 \cdots n] \leq K$$

THE IDEA IS TO DETERMINE FOR EACH ELEMENT A[i] THE NUMBER OF ELEMENTS LESS THAN OR EQUAL TO A[i], WHICH DETERMINES THE PROPER LOCATION FOR A[i].

THE ALGORITHM ALSO DEALS WITH THE SITUATION IN WHICH SEVERAL ELEMENTS HAVE THE SAME VALUE.

CountingSort uses several arrays.

- A[1..n] input array
- B[1..n] output array
- C[0...k] temporary storage (local)

CountingSort(A, B, k) (Pre: $0 \le A[1..n] \le k$, integers, $n = len[A]$)

```
1.) for i ← 0 to k
2.)     C[i] ← 0
3.) for i ← 1 to n
4.)     C[A[i]] ← C[A[i]] + 1
5.) for i ← 2 to k
6.)     C[i] ← C[i] + C[i-1]
7.) for i ← n to 1
8.)     B[C[A[i]]] ← A[i]
9.)     C[A[i]] ← C[A[i]] - 1
```

- loop 1-2 initializes C to all 0's
- loop 3-4 sets C[i] to the number of elements in A which are equal to i
- loop 5-6 sets C[i] to the number of elements in A which are less than or equal to i
- loop 7-9 places elements of A into correct output positions.

OBSERVE THAT IF A CONTAINS DISTINCT ENTRIES, THEN C[A[i]], BEING THE NUMBER OF ELEMENTS LESS THAN OR EQUAL TO A[i], IS THE CORRECT INDEX POSITION FOR A[i]. LINE 8 PLACES IT THERE. IF A CONTAINS REPEATED ELEMENTS, THEN LINE 9 ENSURES THAT ANY ELEMENTS EQUAL TO A[i] WHICH ARE LEFT OF A[i] IN THE ORIGINAL ARRAY, WILL BE PLACED TO THE LEFT OF A[i] IN THE OUTPUT ARRAY.

NOTE THAT CountingSort is <u>STABLE</u> IN THE SENSE THAT ELEMENTS OF THE SAME VALUE OCCUR IN THE OUTPUT ARRAY IN THE SAME ORDER THEY OCCUR IN THE INPUT ARRAY. (THIS IS ONLY RELEVANT WHEN ELEMENTS CONTAIN SATELITE DATA.)

LET $T(n)$ BE THE NUMBER OF ASSIGNMENT OPERATIONS ($\leftarrow$) PERFORMED BY CountingSort. THEN

$$T(n) = k + n + (k-1) + 2n = 3n + 2k - 1$$

$$\therefore \quad T(n) = \Theta(n+k) \; .$$

IF $K = O(n)$, AS IS OFTEN THE CASE IN PRACTICE, THEN $T(n) = \Theta(n)$.

HOW DID WE BEAT THE LOWER BOUND $\Omega(n \lg n)$? CountingSort IS NOT A COMPARISON SORT. IN FACT NO COMPARISONS ARE PERFORMED. INSTEAD THE FACT THAT A CONSISTS OF INTEGERS ONLY IS EXPLOITED.

RadixSort

THIS ALGORITHM ASSUMES THAT EACH ELEMENT OF $A[1..n]$ IS A d-DIGIT NUMBER.

$$A[i] = x_d x_{d-1} \cdots x_3 x_2 x_1$$

x_d: MOST SIGNIFICANT; x_1: LEAST SIGNIFICANT

NOTE: ALTHOUGH IT IS OFTEN THE CASE, WE NEED NOT ASSUME THAT EACH DIGIT IS IN THE SAME RANGE (e.g. 0 - 9).

e.g.

$A[i] =$ YEAR/MONTH/DAY

OR

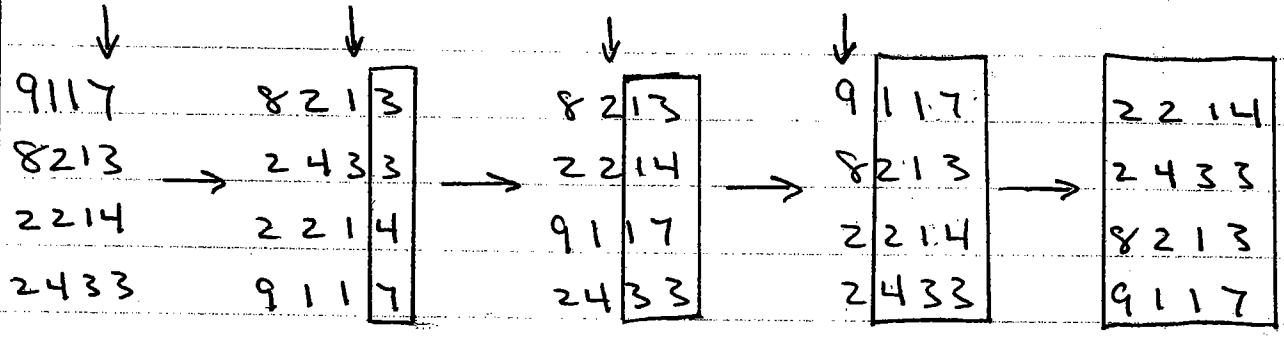
$A[i] =$ A-Z / 0-9 / A-Z

THUS RADIXSORT IN SOME SENSE ALPHABETIZES THE ELEMENTS OF A. WE MAY THINK OF THESE ELEMENTS AS DATA RECORDS, WITH THE "DIGITS" BEING FIELDS.

RadixSort(A,d) (Pre: A CONSISTS OF d-DIGIT NUMBERS)
1.) for i ← 1 TO d
2.) SORT A ON DIGIT i USING A STABLE SORT.

NOTE: IT IS NECESSARY THAT WE SORT FROM LEAST TO MOST SIGNIFICANT DIGIT.

Ex. d = 4



EXERCISE

PROVE THE CORRECTNESS OF RadixSort BY (FINITE) INDUCTION ON i, THE COLUMN BEING SORTED.

INDUCTION HYPOTHESIS : THE LEFT TRUNCATED NUMBERS

$$x_d \cdots x_{i+1} \boxed{x_i \cdots x_1}$$

(i.e. DIGITS 1 TO i) ARE SORTED AFTER THE i^{TH} ITERATION OF LOOP 1-2.

NOTE THE STABILITY OF THE SORT IN (2) IS NECESSARY FOR THE INDUCTION STEP TO WORK.

IF EACH DIGIT IS IN THE RANGE 0 TO (K-1) (i.e. BASE (OR RADIX) K), AND WE USE CountingSort IN (2), THEN RadixSort RUNS IN TIME

$$\Theta(d(n+k))$$

IF $d = \Theta(1)$ AND $k = O(n)$, THIS IS $\Theta(n)$.
IF $k = b$ IS FIXED, THEN RunTime $= \Theta(dn)$
HERE $d = \lfloor \log_b(\max A) \rfloor + 1$.

<u>BucketSort</u>

THIS ALGORITHM ASSUMES THAT THE ARRAY ELEMENTS A[1..n] ARE UNIFORMLY DISTRIBUTED OVER THE INTERVAL $[0, 1)$.

WE DIVIDE $[0,1)$ INTO n SUBINTERVALS OF EQUAL SIZE:

$$\left[0, \frac{1}{n}\right), \left[\frac{1}{n}, \frac{2}{n}\right), \ldots, \left[\frac{n-1}{n}, 1\right).$$

THE UNIFORM DISTRIBUTION ASSUMPTION IMPLIES THAT EACH SUBINTERVAL CONTAINS, ON AVERAGE, 1 ARRAY ELEMENT.

BucketSort USES AN ARRAY $B[1..n]$ OF LINKED LISTS (i.e. BUCKETS.) ITS OUTPUT IS A SINGLE LIST OBTAINED BY CONCATENATING $B[1], \ldots, B[n]$.

BucketSort (A) (Pre: $A[1..n]$ UNIFORMLY DIST. OVER $[0,1)$)

1.) $n \leftarrow$ length[A]
2.) for $i \leftarrow 1$ TO n
3.) insert $A[i]$ INTO list $B[\lfloor nA[i] \rfloor + 1]$
4.) for $i \leftarrow 1$ TO n
5.) sort list $B[i]$ USING InsertionSort
6.) CONCATENATE lists $B[1], \ldots, B[n]$.
7.) return NEW list.

OBSERVE THAT THE FUNCTION $x \to nx$ MAPS

$$[0,1) \longrightarrow [0,n)$$

AND HENCE THE SUBINTERVALS

$$\left[0, \tfrac{1}{n}\right) \longrightarrow [0,1)$$

$$\left[\tfrac{1}{n}, \tfrac{2}{n}\right) \longrightarrow [1,2)$$

$$\vdots$$

$$\left[\tfrac{i-1}{n}, \tfrac{i}{n}\right) \longrightarrow [i-1, i)$$

$$\vdots$$

$$\left[\tfrac{n-1}{n}, 1\right) \longrightarrow [n-1, n)$$

THUS $x \in \left[\tfrac{i-1}{n}, \tfrac{i}{n}\right)$ IS MAPPED TO $nx \in [i-1, i)$, $\therefore \lfloor nx \rfloor = i-1$, $\therefore \lfloor nx \rfloor + 1 = i$.

THUS ALL ELEMENTS IN THE i^{TH} SUBINTERVAL ARE PLACED IN THE i^{TH} BUCKET $B[i]$

THE COST OF ALL OPERATIONS OTHER THAN (5) IS OBVIOUSLY $\Theta(n)$.

IF THERE ARE n_j ELEMENTS IN BUCKET j THEN THE ~~AVERAGE~~ RUN TIME OF BUCKETSORT IS

$$t(n) = \Theta(n) + \sum_{j=1}^{n} \Theta(n_j^2) .$$

OUR ASSUMPTION IMPLIES THAT ON AVERAGE, EACH $n_i = 1$. THUS $\sum_{j=1}^{n} \Theta(n_j^2) = \Theta(n)$, WHENCE

$$t(n) = \Theta(n) .$$

(SEE TEXT FOR MORE RIGOROUS TREATMENT.)

Discrete Bucket Sort

This algorithm takes two inputs: Q and K, where K is a non-negative integer, and Q is a FIFO queue consisting of integers in the range $0 \le x \le K$. It uses an array

$$R[0 \ldots K]$$

of queues for local storage

DiscBucketSort(Q, K)

1.) while $Q \neq \phi$
2.) $\quad x \leftarrow$ Dequeue(Q)
3.) $\quad$ Enqueue(x, $R[x]$)
4.) $Q \leftarrow$ concat($R[0]$, ..., $R[K]$)

Observe that DiscBucketSort is a stable, non-comparison sort. (Exercise)

$$\#\text{QueueOps} = 2n + K = \Theta(n+K)$$

We can use DiscBucketSort as the stable sort in RadixSort.

THE FOLLOWING VERSION OF RadixSort TAKES TWO INPUTS: Q, d, WHERE d IS A NON-NEGATIVE INTEGER AND Q IS A FIFO QUEUE CONSISTING OF d-DIGIT (NON-NEGATIVE) INTEGERS.

WE ASSUME EACH $x \in Q$ IS EXPRESSED IN BASE (OR RADIX) b. WE WRITE $x[i]$ FOR THE i^{TH} DIGIT OF $x \in Q$. THUS FOR EACH i IN THE RANGE $1 \le i \le d$:

$$0 \le x[i] \le b-1$$

WE ALSO USE AN ARRAY $R[0 \cdots (b-1)]$ FOR TEMPORARY STORAGE.

RadixSort(Q, d)

1.) for $i \leftarrow 1$ TO d
2.) while $Q \neq \emptyset$
3.) $x \leftarrow$ Dequeue(Q)
4.) Enqueue$(x, R[x[i]])$
5.) $Q \leftarrow$ Concat$(R[0], \cdots, R[b-1])$

Queue OPS $= d(2n+b-1) = \Theta(dn)$

AGAIN $d = \lfloor \log_b(\max Q) \rfloor + 1$