8: Sorting in Linear Time

Theorem: Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

* **Proof:** → $l \geq n!$

→ By Lemma: $n! \leq l \leq 2^h$ or $2^h \geq n!$

Take log both side $h \geq \lg(n!)$

Use Stirling approximation: $n! > (n/e)^n$

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \end{aligned}$$

* Proof by induction on h.

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$

Inductive step: Assume true for height $= h-1$.

Extend tree of height $h-1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

# of leaves for height $h = 2 \cdot$ (# of leaves for height $h-1$)

$= 2 \cdot 2^{h-1}$

$= 2^h$

Exercises:

8.1-1 What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

For a permutation $a_1 \leq a_2 \leq \cdots \leq a_n$, there are $n-1$ pairs of relative ordering, Thus the smallest possible depth is $n-1$.

8.1-2: obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

$$\sum_{k=1}^{n} \lg k \leq \sum_{k=1}^{n} \lg n = n \lg n.$$

$$\sum_{k=1}^{n} \lg k = \sum_{k=2}^{n/2} \lg k + \sum_{k=n/2}^{n} \lg k$$

$$\geq \sum_{k=1}^{n/2} 1 + \sum_{k=n/2}^{n} \lg n/2$$

$$= \frac{n}{2} + \frac{n}{2}(\lg n - 1)$$

$$= \frac{n}{2} \lg n$$

8.1-3 Show that there is no comparison sort whose running time is linear for at least half of the $n!$ input of length n. What about a fraction of $1/n$ of the I/ps of length n? What about a fraction $1/2^n$?

Consider a decision tree of height h with ℓ reachable leaves corresponding to a comparison sort on n elements.

i.e. $\frac{n!}{2} \leq n! \leq \ell \leq 2^h$

Taking logs

$$h \geq \lg(n!/2) = \lg(n!) - 1 = \Theta(n \lg n) + \cdots$$

$$\boxed{h = \Theta(n \lg n).}$$

From the equation above, there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n.

Consider the $1/n$ of inputs of length n condition. We have $(1/n)\, n! \leq n! \leq \ell \leq 2^h$. By taking logs, we have

$$h \geq \lg(n!/n) = \lg(n!) - \lg n = \Theta(n \lg n) - \lg(n)$$

$$\boxed{h = \Theta(n \lg n)}$$

From the equat^n above, there is no comparison sort whose running time is linear for $1/n$ of the $n!$ inputs of length n.

Consider the $1/2^n$ of inputs of length n condition,

We have $(1/2^n)\, n! \le n! \le \alpha \le 2^h$. By taking logs

$$h \ge \lg(n!/2^n) = \lg(n!) - n = \Theta(n \lg n) - n$$

$$\boxed{h = \Theta(n \lg n)}$$

From the equation above, there is no comparison sort whose running time is linear for $1/2^n$ of the $n!$ inputs of length n.

8.1-4: Assume that we need to construct a binary decision

Ans: tree to represent comparisons. Since length of each subsequence is k, there are $(k!)^{n/k}$ possible o/p permutations. To compute the height h of the decision tree, we must have $(k!)^{n/k} \le 2^h$. Taking logs on both sides, we know that

$$h \ge \frac{n}{k} * \lg(k!) \ge \frac{n}{k} * \left(\frac{n \ln k - k}{\ln 2}\right) = \frac{n \ln k - n}{\ln 2}$$

$$\boxed{h = \Omega(n \lg k)}$$ Ans.

8.2: Counting Sort

Depends on a key assumption: numbers to be sorted are integers in $\{0, 1, \dots, k\}$.

COUNTING-SORT (A, B, n, k)

```
(1) for i = 0 to k                  ——— k times  Θ(k)
(2)     do C[i] = 0
(3) for j = 1 to n                  ——→ n times  Θ(n)
(4)     do C[A[j]] = C[A[j]]+1

(5) for i = 1 to k                  ——→ k times = Θ(k)
(6)     do C[i] = C[i] + C[i-1]

(7) for j = n down to 1  ——→  Θ(n).  }
(8)     do B[C[A[j]]] = A[j]          }  (1 to 12)
(9)        C[A[j]] = C[A[j]] - 1      }
```

Ex:- Figure 8.2

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

(a) C — C[i-1]

0	1	2	3	4	5
2	0	2	3	0	1

(b) C[i]

0	1	2	3	4	5
2	2	4	~~7~~ 7	7	8

(c) B

1	2	3	4	5	6	7	8
						3	

C

0	1	2	3	4	5
2	2	4	6	7	8

(d)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

→ Final B

	0	1	2	3	4	5
C	~~1~~ 0	2	~~4~~ ~~3~~ 2	~~6~~ ~~5~~ 4	7	~~8~~ 7

	0	1	2	3	4	5
C	0	2	2	4	7	7

* Counting sort is stable (keys with same value appear in the same order in o/p as they did in input) because of how the last loop works.

Analysis: $\Theta(n+k)$, which is $\Theta(n)$ if $k = O(n)$.

Exercise:

8.2-1: $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

(a)

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

C[i-1]

(b)

	0	1	2	3	4	5	6
C	2	4	6	8	9	9	11

C[i]

(c) [n to 1]

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

	0	1	2	3	4	5	6
C	~~2~~ ~~1~~ 0	~~4~~ ~~3~~ 2	~~6~~ ~~5~~ 4	~~8~~ ~~7~~ 6	~~9~~ 8	9	~~11~~ ~~10~~ 9

Ans.

8.2-2: Prove that COUNTING-SORT is stable.

Suppose positions i & j with i<j both contain some element k. We consider lines 10 through 12 of counting-sort, where we construct the o/p array. Since j>i, the loop will examine A[j] before examining A[i]. When it does so, the algorithm correctly places A[j] in position m = C[k] of B. Since C[k] is decremented in line 12 & is never again incremented, we are guaranteed that when the for loop examines A[i] we will have C[k] < m. Therefore A[i] will be placed in an earlier position of the o/p array, proving stability.

for j = n to 1
do B[C[A[j]]] = A[j]
C[A[j]] = C[A[j]] - 1

8.2-3: for j = 1 to n } then it work properly in COUNTING-SORT

The algorithm still works correctly. The order that elements are taken out of C & put into B doesn't affect the placement of the elements with the same key. It will still fill the interval (C[k-1], C[k]) with elements of key k.

We could think of extending this algorithm by placing the element A into collection of elements for each cell in array C. Then, if we use a FIFO collection, the modification of

line 10 will make it stable, if we use LITO; it will be anti-stable.

8.2-4 Ans. The algorithm will begin by preprocessing exactly as COUNTING-SORT does in Line 1 through 9. So that C[i] contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range [a..b], simply compute C[b] - C[a-1]. This takes O(1) times & yields the desired o/p.

8.3: Radix Sort

Key idea: Sort least significant digits first.

RADIX-SORT(A,d)

① for i ← 1 to d

② do use a stable sort to sort array A on digit i.

Analysis: Assume that we use counting sort as the intermediate sort.

→ θ(n+k) per pass (digits in range 0,->,k)

→ d passes

→ θ(d(n+k)) total.

→ If k = O(n), then time = θ(dn).

Exercise

8.3.2 Stable sort: Insertion sort & Merge sort.

Non-stable sort: Heapsort & quicksort

Scheme of making any sorting algorithm stable, we can preprocess, replacing each element of an array with an ordered pair. The first entry will be the value of the element & the 2nd value will be the index of the element.

Ex:- A[2, 1.1, 3, 4, 4, 4] would become

[(2,1), (1,2), (1,3), (3,4), (4,5), (4,6), (4,7)]. We now interpret (i,j) < (k,m) if i < k or i=k & j<m. Under this definition of less than, the algorithm is guaranteed to be stable because each of our new elements is distinct & the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space requirement, but the running time will be asymptotically unchanged.

8.3-3: Loop invariant: At the begining of for loop, the array is sorted on the last i-1 digits.

Initialization: The array is trivially sorted on the last 0 digits.

Maintenance: Let's assume that the array is sorted on the last $i-1$ digits. After we sort on the i^{th} digit, the array will be sorted on the last i digits. It is obvious that elements with different digit in the i^{th} position are ordered accordingly. In the case of the same i^{th} digit, we still get a correct order, because we are using stable sort if the elements were already sorted on the last $i-1$ digits.

Termination: The loop terminates when $i = d+1$. Since the invariant holds. We have the numbers sorted on d digits.

8.3-4: Show that to sort n integers in the range 0 to n^3-1 in $O(n)$ time.

First run through the list of itegeres & convert each one to base n, then radix sort them. Each number will have at most $\log_n n^3 = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we use counting sort to sort each digit in $O(n)$ time.

***8.3-5** Given n d-digit numbers in which each digit can take on up to k possible values, we'll perform $O(kd)$ passes & keep track of $\theta(nk)$ piles in the worst case.

Lemma: Given n b-bit numbers & any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\theta((b/r)(n+2^r))$ time if the stable sort it uses takes $\theta(n+k)$ time for inputs in the range 0 to k.

Proof: For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$. So, that we can use counting sort with $k = 2^r - 1$. (For example we can view a 32-bit word as having four 8-bits digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ & $d = b/r = 4$) Each pass of counting sort takes time $\theta(n+k) = \theta(n+2^r)$ & there are d passes, for a total running time of $\theta(d(n+2^r)) = \theta((b/r)(n+2^r))$.

8.4 : Bucket Sort

Assume the input is generated by a random process that distributes elements uniformly over [0, 1).

Idea:

→ Divide [0, 1) into n equal-sized buckets

→ Distribute the n i/p ~~variable~~ values into the buckets.

→ Sort each bucket.

→ Then go through buckets in order, listing elements in each one.

```
BUCKET-SORT(A)
① let B[0 -- n-1] be an array
② n = A.length
③ for i = 0 to n-1
④     make B[i] an empty list
⑤ for i = 1 to n
⑥     insert A[i] into list B[⌊n A[i]⌋]
⑦ for i = 0 to n-1
⑧     sort list B[i] with insertion sort.
⑨ Concatenate the lists B[0], B[1] -- B[n-1] together in order.
```

Correctness: Consider $A[i] \neq A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Then $\lfloor n \cdot A[i] \rfloor \le \lfloor n \cdot A[j] \rfloor$. So $A[i]$ is placed into the same bucket as $A[j]$ or into a bucket with a lower index.

→ If the same bucket, insertion sort fixes up.

→ If earlier bucket, concatenation of lists fixed up.

Analysis:

→ Relies on no bucket getting too many values.

→ All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.

→ Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow O(n)$ sort time for all buckets.

→ We "expect" each bucket to have few elements, since the average is 1 element per bucket.

→ But we need to do a careful analysis.

Define a random variable:

n_i = the number of elements placed in bucket $B[i]$.

Because insertion sort run in quadratic time, bucket sort time is

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

Take expectations of both sides:

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

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \quad \text{(linearity of expectation)}$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad (E[aX] = aE[X])$$

Claim: $E[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

→ $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$

→ $Pr\{A[j] \text{ falls in bucket } i\} = 1/n$

→ $n_i = \sum_{j=1}^{n} X_{ij}$

Then

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij} X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} E[X_{ij} X_{ik}] \quad \text{[Linearity of expectation]}$$

$E[X_{ij}^2] = 0^2 \cdot Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot Pr\{A[j] \text{ falls in bucket } i\}$

$= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n}$

$= 1/n$

$E[X_{ij} X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} & X_{ik} are independent random variables

$\Rightarrow E[X_{ij} X_{ik}] = E[X_{ij}]\, E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = 1/n^2$

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + 2\binom{n}{2} \cdot \frac{1}{n^2}$$

$$= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n}$$

$$= 1 + 1 - \frac{1}{n}$$

$$\boxed{E[n_i^2] = 2 - 1/n}$$ Hence proved claim.

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

$$= \Theta(n) + O(n)$$

$$= \Theta(n)$$

→ Again, not a comparison sort. Used a function of key values to index into an array.

→ This is a probabilistic analysis — we used probability to analyze an algorithm whose running time depends on the distribution of inputs.

→ Different from a randomized algorithm, where we use randomization to impose a distribution.

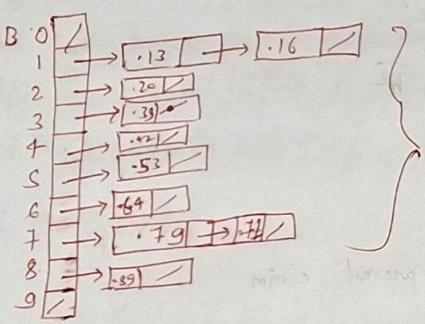
→ With bucket sort, if the input isn't drawn from a uniform distribution on $[0, 1)$, all bets are off (performance-wise, but the algorithm is still correct).

Exercise

B[⌊nA[i]⌋]

8.4-1: $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$

$A[1 \ldots 10]$ & $B[0 \ldots 9]$



$A = \langle .13, .16, .20, .39, .42, .53, .64, .71, .79, .89 \rangle$

8.4-2: Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time & makes its worst-case running time $O(n \lg n)$?

If all the keys fall in the same bucket & they happen to be in reverse order, we have to sort a single bucket with n items in reversed order with insertion sort. This is $\Theta(n^2)$.

We can use merge sort or heapsort to improve the worst-case running time. Insertion sort was chosen because it operates well on linked lists, which has optimal time & requires only constant extra space for short linked lists. If we use another sorting algorithm, such as merge sort, instead of insertion sort when sorting the buckets.

8.4-3: Let X be a random variable that is equal to the No. of heads in 2 flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

$$E[X] = 2 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1$$

$$E[X^2] = 4 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1.5$$

$$E^2[X] = E[X] \cdot E[X] = 1 \cdot 1 = 1$$

***8.4-4:**

$$\pi r_i^2 = \frac{i}{n} \cdot \pi \cdot 1^2$$

$$r_i = \sqrt{\frac{i}{n}}$$

Bucket sort by radius.

***8.4-5:** Bucket sort by p_i, so we have n buckets: $[P_0, P_1), [P_1, P_2), \ldots \rightarrow [P_{n-1}, P_n)$. Note that not all buckets are the same size, which is ok as to ensure linear run time, the p_i should on average be uniformly distributed amongst all buckets, of which the intervals defined with p_i will do so.

p_i is defined as follows: $\boxed{P(p_i) = \frac{i}{n}}$