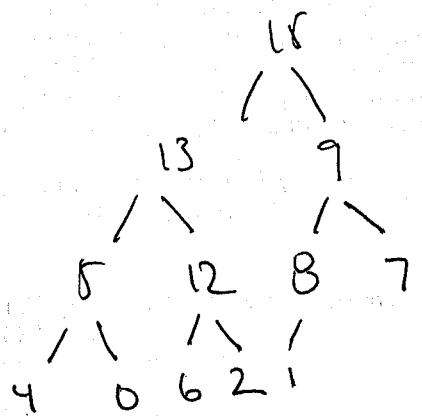
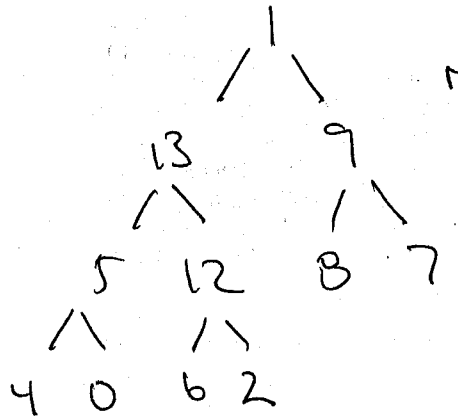
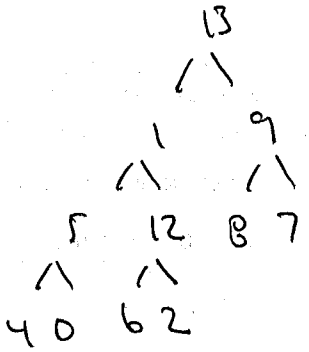
6.5-1



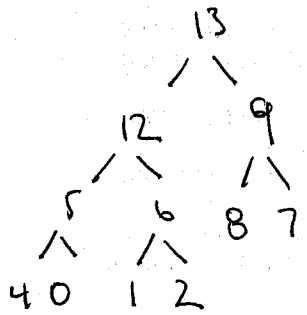
Extract Max



MaxHeapify(A,1)

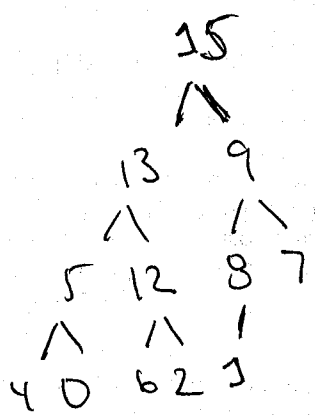


⇒

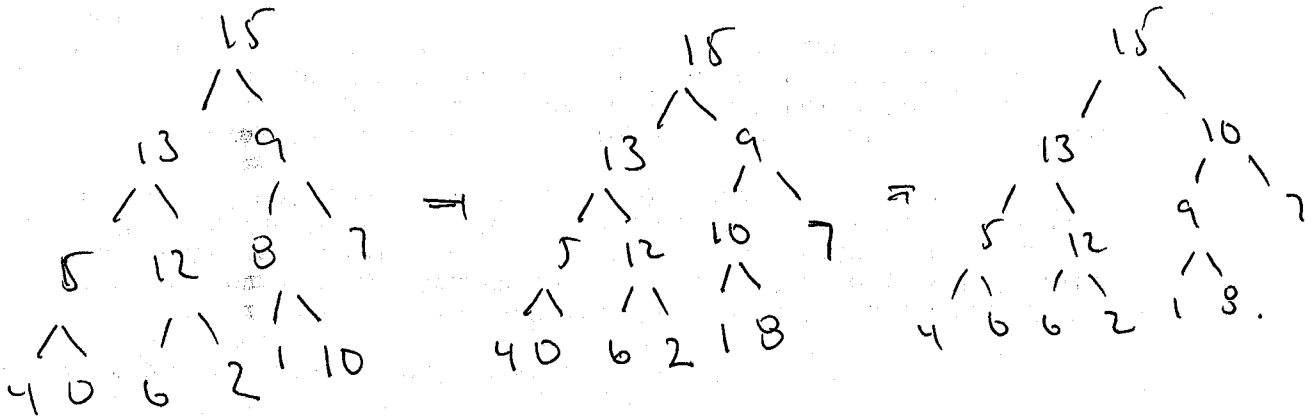


is final heap.

6.5-2



Max-heap insert.



6.5-3

```
Heap-Minimum(A)
    return A(1)
--
Heap-Extract-Min(A)
    if heap-size(A) < 1
        then error 'heap underflow'
    min ← A[1]
    A[1] ← A[heap-size[A]]
    heap-size[A] ← heap-size[A]-1
    Min-Heapify(A,1)
    return min.
```

Heap-Decrease-key (A, i, key)

if key > A[i]

then error "new key is greater than current key"

A[i] ← key

while (i > 1 & A[parent[i]] > A[i]) do

exchange A[parent[i]] & A[i]

i ← Parent[i]

end-do

Min-Heap-insert (A, key)

heap-size(A) ← heap-size(A) + 1

~~A[heap-size[A]] ← key~~

A[heap-size(A)] ← +∞

Heap-Decrease-key [A, heap-size(A), key]

6.5-4 Well if we didn't & the location in memory held a # ~~greater~~ greater than the value of key the call to Heap-increase-key would crash.

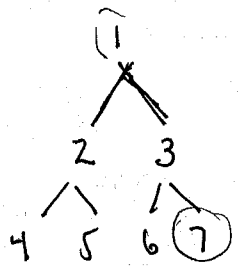
6.5-5 Show this is true ~~before~~, during & after the loop

before: since key must be $\geq A[i]$ to have ~~made~~ made it this far the loop invariant holds.

during: During each iteration ~~the~~ ~~proof~~ here this invariant holds.

After: ?

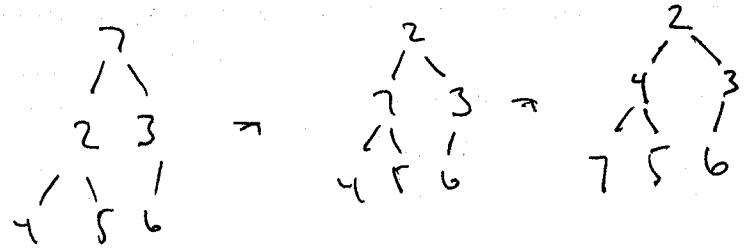
6.5-6 First in First out priority queue. could be implemented as follows



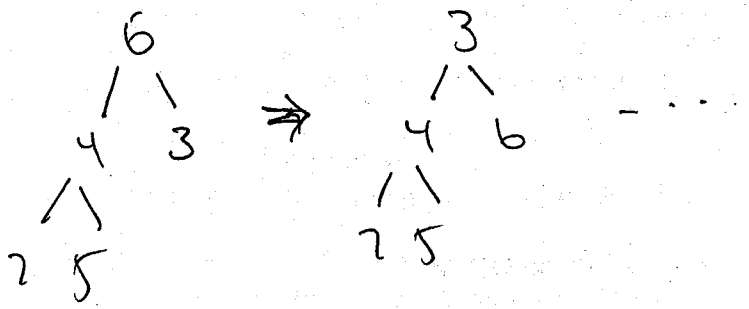
: min-heap

~~6.~~ Implement a min-heap ~~for~~ tagging each input digit w/ a next increasing #. Then to delete items of the F.I.F.O. call

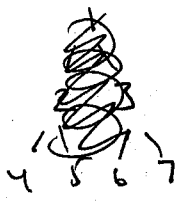
extract-min(A)



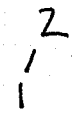
Next extract:



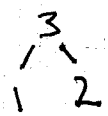
To implement a ~~Stack~~ (L.I.F.O)

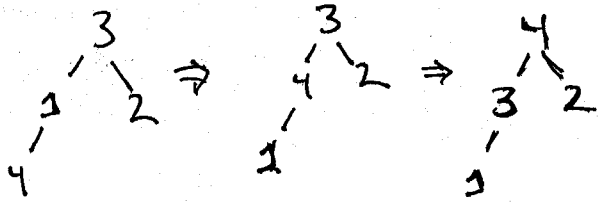


Max-heap



Max-heap





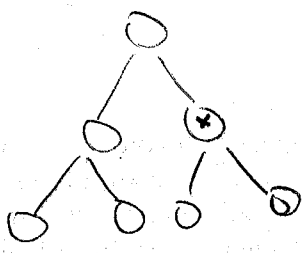
~~For~~

Use a max-heap + for every element entered ~~increment~~ use an increment of the count i.e 1, 2, 3, 4, ...,

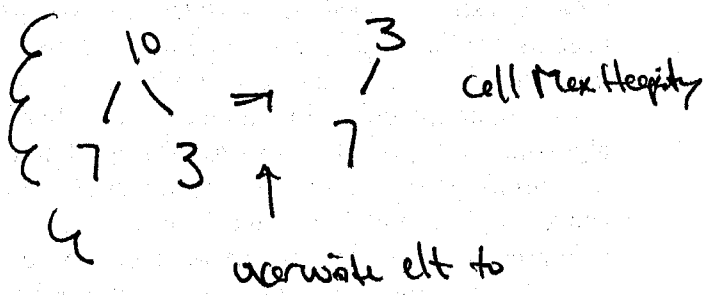
Call Max-heap-insert(A, key) w/ key = 1, 2, 3, ...

Call ~~Heap~~ Heap-Extract-Max(A) + pop elements of

6.5-7



implement Heap-Delete:



Heap-Delete(A,i)

1) Find Right most decendent of Node i say r.

2) Exchange A[r] + A[i]

~~3) call Max-Heapify~~

3) heap-size(A) ← heap-size(A)-1

4) call Max-Heapify(A,i)

This is O(lg n) if I can show Find right most decendent is. (known since it involves calling Right(i) at most lg n times)

Find Rightmost Decendent(i)

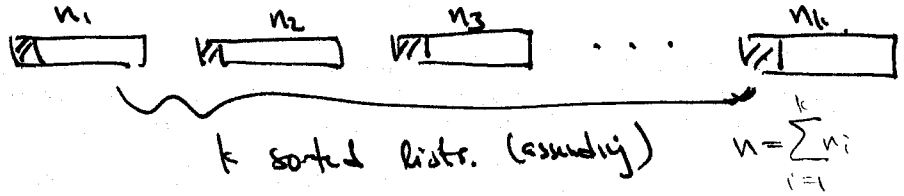
~~while (i < heap-size(A))~~ (right(i) < heap-size(A))

~~i = right(i)~~ i = right(i)

endwhile endwhile.

pg 142 CLR

6.5.8



Note: 1) An ascending sorted list is a min heap.

Following hint:

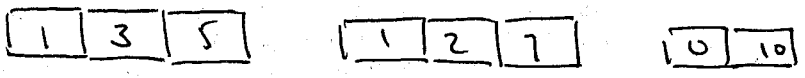
Place k lst elements in a min heap. (place elt (key) + index of list it came from)

Time to construct a heap from k elements is ~~$O(\lg k)$~~ $O(k)$.

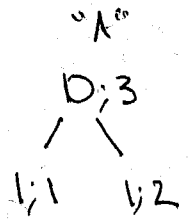
Then repeatedly call

extract min + put in Auxiliary array.
insert

Eg: w/ 3 lists



min heap w/ lst elements



Notation $(k;l)$ = (key, list index)

Then for $i = 1;n$ ~~do~~ do call

1) extract-min from min heap : $O(\lg k)$ place in output array. Learn from what list it came from say l.
2) Min-Heap-Insert$(A, k_{next}(l))$: $O(\lg k)$

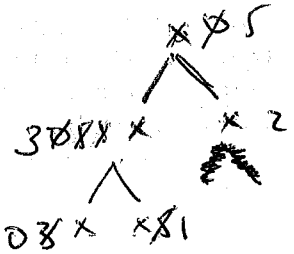
end

$\Rightarrow$ overall time (total) = $O(n \lg k)$

pg 142 CLR

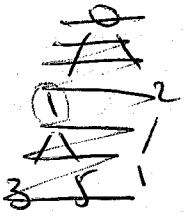
6-1 Build-Max-Heap calls:
{ Max-Heapify bubbles elt i down }
{ Heap-increase-key bubbles elt i up }

(a) let A = 0 1 2 3 5

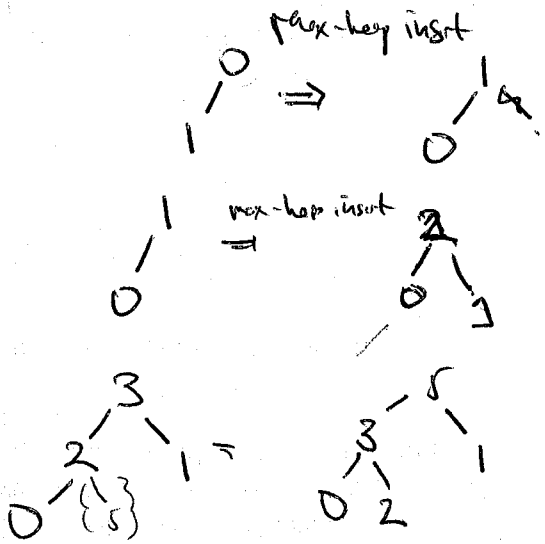


Build-Max-Heap'(A) $\lfloor \frac{5}{2} \rfloor = 2$

produces 5 3 2 0 1



Build-Max-Heap'(A) produces



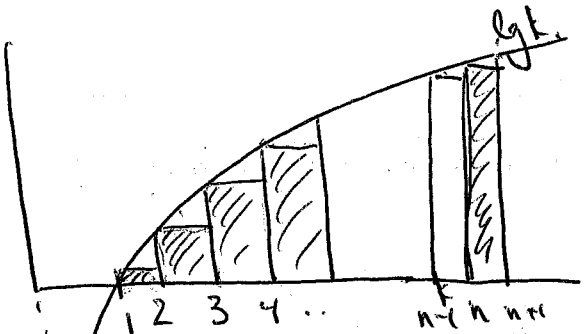
5 3 1 0 2

So the two procedures do not produce the same arrays.

(b) Max-heap insert requires $O(\lg n)$

Thus Build-Max-Heap'(A) can require as a sorted list, sorted incorrectly i.e. backwards to the heap direction.

$$\sum_{k=1}^{n} O(\lg k) = O\left(\sum_{k=1}^{n} \lg k\right)$$



$$\lg k = \frac{\ln k}{\ln 2}$$

$$\int_0^n \lg k\,dk \le \sum_{k=1}^{n} \lg k \le \int_1^{n+1} \lg k\,dk = \frac{1}{\ln 2}\int_1^{n+1} \ln k\,dk$$

$$= \frac{1}{\ln 2}\left[k\ln k - k\right]_{k=1}^{n+1}$$

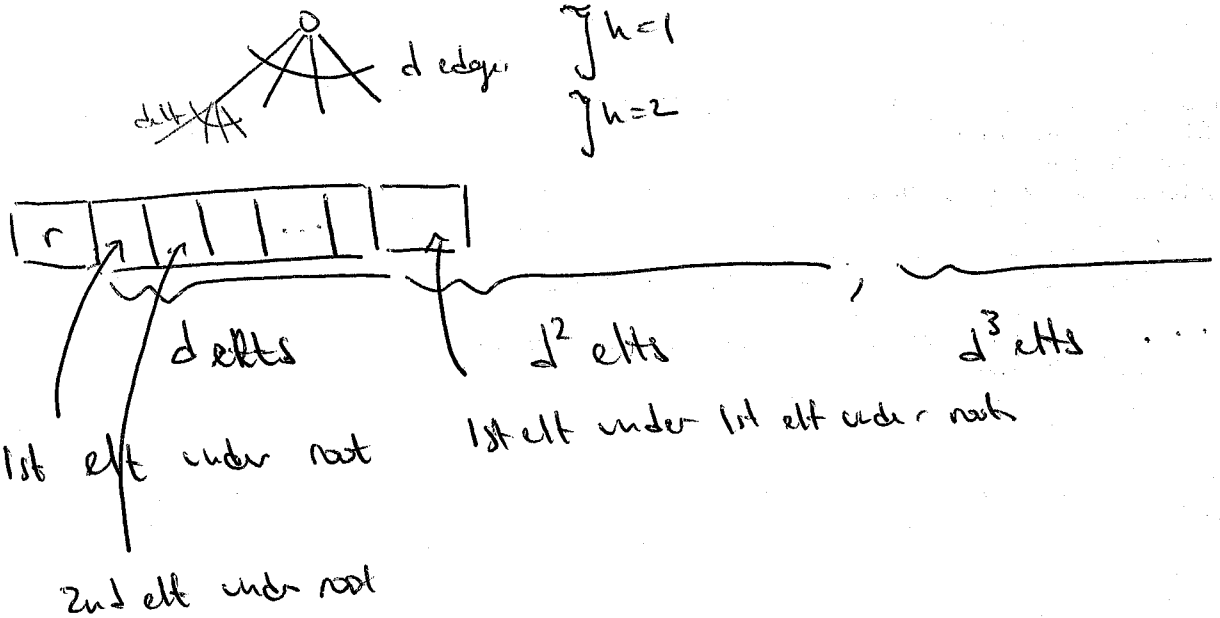
$$= (n+1)\lg(n+1) - \frac{(n+1)}{\ln 2} = \Theta(n\lg n)$$, this bound would work from the other side also

$$\therefore \sum_{k=1}^{n} \lg k = \Theta(n\lg n)$$

∴ ~~Max-Heap~~ Build-Max-Heap' can require $\Theta(n\lg n)$ time.

6-2

(a)



(b) let n = # elts in a d-ary heap

$$n \leq 1 + d + d^2 + \cdots + d^h = \sum_{k=0}^{h} d^k = \frac{d^{h+1}}{d-1}\Big|_0^{h+1}$$

roughly $h = \log_d n$

$$n \leq \frac{d^{h+1} - 1}{d-1}$$

$$\Rightarrow d^{h+1} = (d-1)n + 1$$

$$h = \log_d((d-1)n+1) - 1$$

(c) Extract-Max(A)

max = A[1]

heap-size[A] ← heap-size[A] - 1

i = 1

while (i < heap-size[A])

```
if (left(i) > right(i))
    move A[left(i)] into A[i]
    i := left(i)
else
    move A[right(i)] into A[i]
    i := right(i)
endif
```

not correct because we must store over a d-array subscript for the max

Define child(i,j) $1 \le j \le d$ to be the jth child of the ith node & child(i,0) = i.

Then Extract-max(A) sho

```
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
Max-Heapify(A, 1)
```

& letting max-heapify(A,i) as

```
maxindex = 1
for (j = 1:d)
    if (A[child[i,j]] > A[child[i,maxindex]])
        maxindex = j
endif
```

exchange (A[i] + A[child[i, maxindex]])

call Max-Heapify(A, child

Define Max-Heapify(A,i) for a d-ary heap as

maxindex = 0

All check that we have child nodes, ie we are not on a leaf.

for (j = 1:d)

if (A[child[i, d]] > A[child[i, maxindex]])

then maxindex = j

end if

end for

if (maxindex ≠ 0)

exchange A[i] + A[child[i, maxindex]]

Max-Heapify (A, child[i, maxindex])

end if

Running time $O(d \cdot \log_d n)$.

(e) Increase-key(A, i, k) { need to "bubble up this elt" }

if (k < A[i]) error.

while (i > 1)

~~if percent[i] <~~

if (A[percent[i]] < A[i])

exchange A[percent[i]] & A[i].

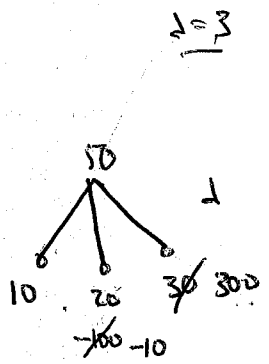
i = percent[i]

else

~~return~~ break

end

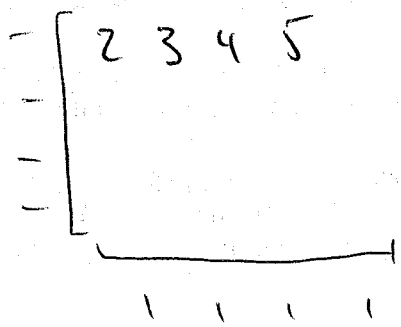
endwhile



The running time of this algo is $O(\log_d i)$ count of traversals till I get to the top of the heap.

6-3

(a)



9, 16, 8, 14, 12

2, 3, 4, 5, 8, 9, 12, 14, 16 is so-

One way:

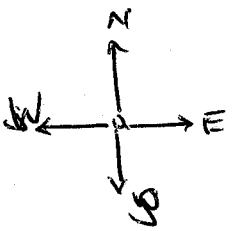
$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 8 & 9 & 12 & 14 \\ 16 & \infty & \infty & \infty \end{bmatrix}$$

another w/ more "space" between elts

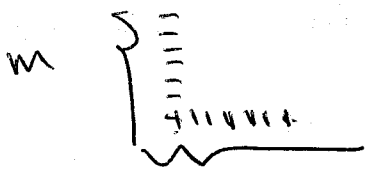
$$\begin{bmatrix} 2 & 3 & \infty & 4 \\ 5 & \infty & 8 & 9 \\ \infty & 12 & 14 & 16 \end{bmatrix}$$

} but think this is a valid young tableau.

(b) These seem like devious sequences since No # is $\geq +\infty$ & every # is $< +\infty$.



(c) ~~Extract-Min~~



define ~~North(i,j)~~ = (

~~North~~ East(i,j) = i, j+1

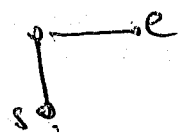
South(i,j) = i+1, j

Extract-min(Y) $Y \in \mathbb{R}^{m \times n}$ Young tableau

if Y(1,1) = +∞ error

e = East(1,1)

S = South(1,1)

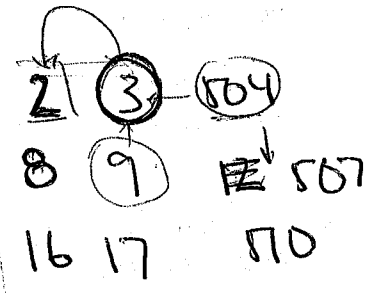


min = Y(1,1)

if (Y(e) < Y(s))

copy Y(e) into Y(1,1)

ReTableau(Y, e, m, n-1)



else

copy Y(s) into Y(1,1)

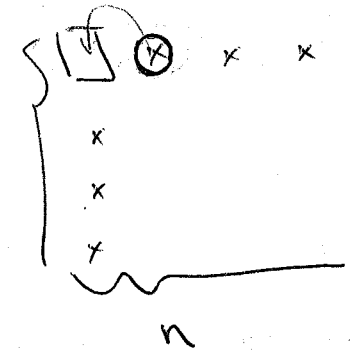
ReTableau(Y, s, m-1, n)

end

ReYoungTableau will take a Young's Tableau of size mxn

Young's

return a new tableau

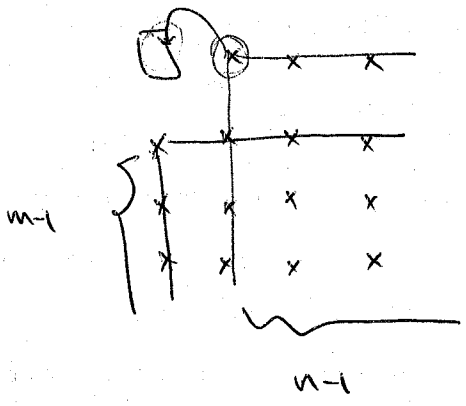


ReYoungTableau will take a Young's tableau m of size mxn except where possibly the (1,1) element is not present. (or invalid to Young's tableau form)

ReYangTableau(Y, i, j, m, n) $Y \in \mathbb{R}^{m \times n}$

if (Y(i, j+1) < Y(i+1, j)) ← if

exchange



... I have the correct idea but...

$$T(p) = C + T(p-1) \Rightarrow T(p) = C + C + T(p-2)$$

$$= 3C + T(p-3)$$

$$= p \cdot C + T(0)$$

$$\Rightarrow T(p) = (n+m) \cdot C + T(0)$$

$$T(p) = O(n+m)$$

(d) i.e. put 6 into initial gin Kang Taskeen

Idea: put elt in 1st +ve location, then bubble it up word until it reaches its natural level. This is done in the following

look to the elt above & to the left of the location of the new elt. Exchange the new elt w/ the smaller of the two elts & move the point of focus to the new location of the elt. Stop when neither the ~~upper~~ upper (from above) or left elts are larger than the current elt or the current elt is in the (1,1) location.

This would take $O(n+m)$ time since ~~we~~ ~~could~~ ~~encounter~~, at most this would be the # of elts searched are

(e) Think of two pointers on two elts & constantly return the smaller of the two & moves the

Maybe like this: for a given pt. return the minimum of the pair to the right & below. If you return the pair below

↳ move your pointer to that point & repeat

If you move your pointer to the right, move your ptr

then and repeat. If the pointer gets to the bottom of a column or end of a row move to the next column or next row.

(if) Maybe insert a special # & see it it ends up in the correct location like -∞ & see if it ends up at location (1,1)?

?

8.1-1

pg 155 CLR

$A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$

$x = 13$

6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21

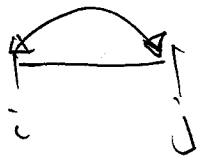
6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21

pg 148 CLR

2-1-1

6.1-2

pg 155 CLR



each time through while loop we decrement $i+j$ until $i+j = $ ~~6~~ $\Rightarrow i=j$

$i=1, \quad j=3$ Part(A, 1, 3)

$i=0 \quad j=4$

$j=3 \quad i=1$

$j=2 \quad i=2.$

Part(A, 1, 4)

$i=0 \quad j=5$

$j=4 \quad i=1$

$j=3 \quad i=2$

$j=2 \quad i=3 \leftarrow$ ret j

pg 148 CLR

Part(A, p, r) return $q = \left\lfloor \frac{p+r}{2} \right\rfloor$ if All ~~the~~ elts are the same

6.1-2

8.1-3 We must access every element to see if it is $\leq$ the key that we are pivoting around $\therefore \Omega(n)$

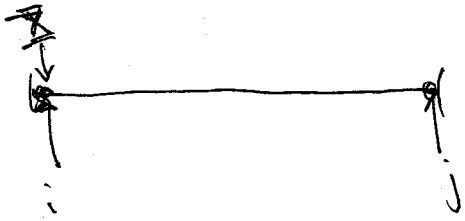
But we also don't do any other operations besides those so $O(n)$

$\rightarrow \Theta(n)$

8.1-4 Change lines 6 + 8 in Partition to read

until $A[j] \geq x$

until $A[i] \leq x$



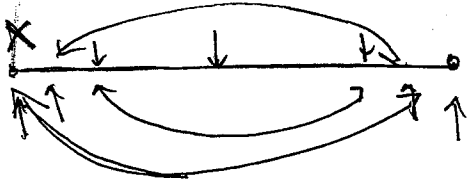
148 CLR

7.1-3 + 7.1-5

8.2-1

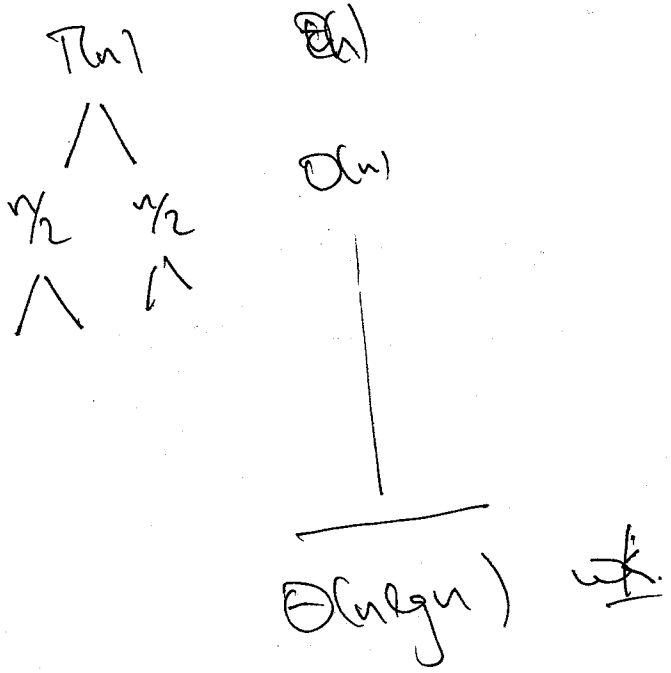
Pg 160 CLR

$T(n) = \Theta(n) + T(n-1)$



we ÷ ½ each time Thus
(q ends up by the middle value)

$T(n) = \Theta(n) + 2T(n/2)$



Pg 153 CLR

7.2-1

7.2-2 If all the elts of Array A have the same value the procedure would run better than insertion sort since the 2 partitions would be of size $n-1$ & 1 at each step.

$\therefore \quad T(n) = \Theta(n^2)$

Pg 160 CLR

8.2-2 We basically have to show that ~~[illegible]~~ unnecessary order elicits an unbalanced partition at each step.

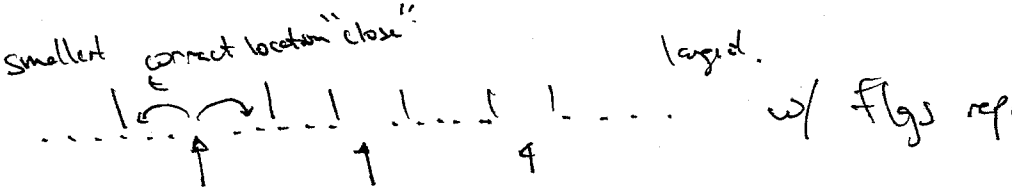
7.2-3 But as we are partitioning around the largest elt in the list ~~[illegible]~~ the only elt ≥ this "pivot" is itself & thus a list must contain n-1 values.

This can be seen to occure at every level

$\Rightarrow T(n) = \Theta(n^2)$ By Pg 156 CLR

8.2-3 Pg 153 CLR

input of type



w/ flgs repeat elts at of order.

7.2-4 Quick sort would look at the 1st elt & then go elt to elt looking for elements to move in the partition shown as

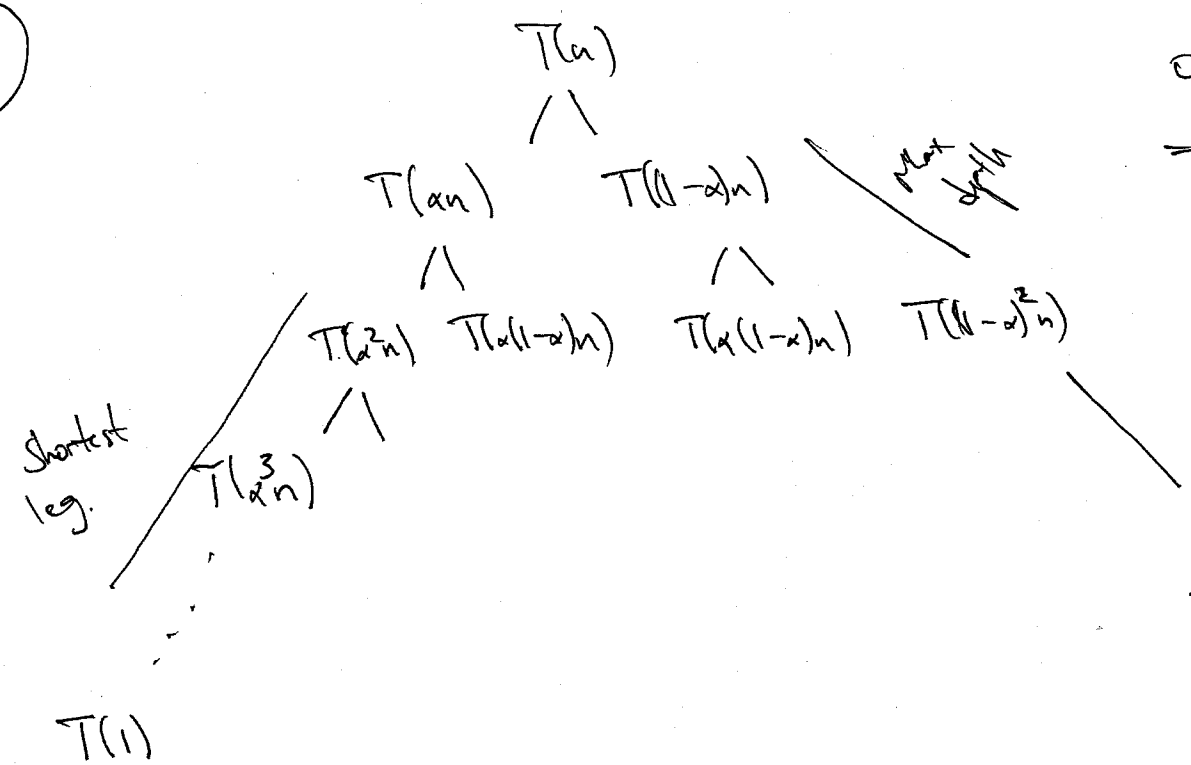
Assuming 1st few are in order ~~having a sorted sequence causes QS to partition 1, n-1.~~

~~But After the partition we have made no~~ Q.S. would partition 1, n-1 (worst case) every time. Until it got to w/in ε of the ~~[illegible]~~ misplaced item Thus after ~~[illegible]~~ several calls to QS we begin to make progress w these partitions.

Insertion just runs down the list moving each elt to its correct location

Ass each misplaced elt does at how far to go. I.S is best.

8.2-4



$0 < \alpha \leq 1/2$

$\Rightarrow \quad 1-\alpha > \alpha.$

Thus

p.253 CLR

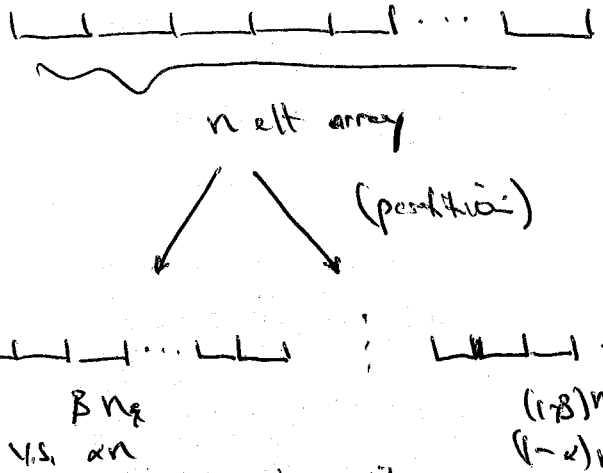
5.2.5

$\alpha^p n \doteq 1$

$p \doteq \log_{\alpha} 1/n = \dfrac{-\lg(n)}{\lg \alpha}.$

$(1-\alpha)^p \doteq 1$

7.2.6



Given α,
$0 < \alpha \leq 1/2$

What is the probability that the split produces a more balanced array than $1-\alpha$ & α?

more balanced means that $\beta \geq \alpha$.

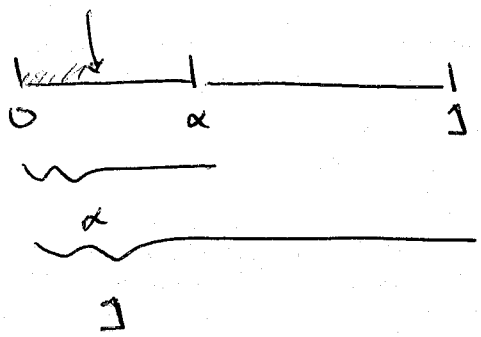
Thus P(Partition produces a split more balanced than αn & $(1-\alpha)n$)

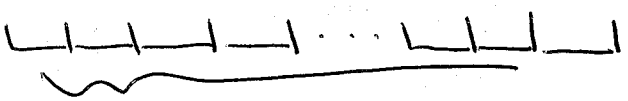
$= 1 -$ P(Partition produces a split less balanced than αn & $(1-\alpha)n$)

Now:
Prob(Partition produces a split less balanced than αn & $(1-\alpha)n$)

= Why is this equal 2α?

If I can show that ... I am done.

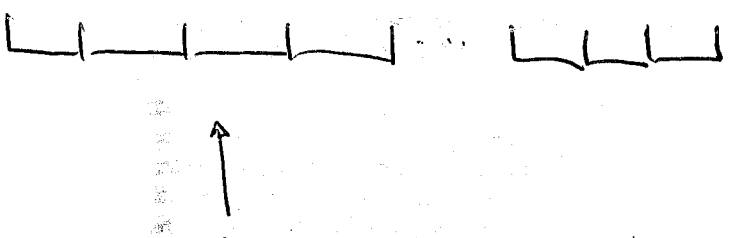




n #'s. What is the prob that I will pick a # s
when pivoted that it will produce a left 1/2 sub array

or a right 1/2 sub array of size $< \alpha$.

Consider the left 1/2 sub array. Consider the n elts sorted. Then it becomes clear that



up to αn elts or w/ probability α. will I choose from among these αn elts.

But this same choice would be made from the top also (i.e. I could make a small partition at the top.)

∴ Prob That a worse ~~split~~ partition than α occurs is $\leq (1-\alpha)$ across 1

2α

∴ Prob split more balanced $= 1 - 2\alpha$ ✓.

(7.3-1) Worst case performance will not be able to happen repeatedly, while random case performance will.

(7.3-2) In worst case

N = # calls to random # generator $N = N(n)$ n = length of input string

$$N(n) = 1 + N(q-1-p+1) + N(r-q-1+1)$$

↑ in call to Randomize-Partition(A,p,r)

$\Rightarrow N(n) = 1 + N(q-p) + N(r-q)$

$n = r-p+1$

$r = n+p-1$

$$N(n) = 1 + N(q-p) + N(n+p-1-q)$$

$$= 1 + N(q-p) + N(n-(q-p)-1)$$

In the worst case the partitions are of sizes 1 & $n-1$

$\Rightarrow N(n) = 1 + \overset{=0}{N(1)} + N(n-1) \Rightarrow N(n) = 1 + N(n-1)$

$\Rightarrow N(n) = \Theta(n)$

In last case there should be no difference.

Pg 188 CLR

08-06-02

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

Guess $T(n) \le cn^2$

$$T(n) \le (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$\le c$$

$q = 0 \Rightarrow T(0) + T(n-1)$

$q = n-1 \Rightarrow T(n-1) + T(n-n+1-1) = T(0)$

Why range of q is from 0 to $n-1$?

$$T(n) \le cn^2 - c(2n-1) + \Theta(n) \le cn^2$$

Pg 156 CLR

$X_{ij} = I\{z_i \text{ is compared to } z_j\}$.

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum\sum$$

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

$$\sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1}$$

$$= \sum_{i=1}^{n} O(\lg n)$$

$$= O(n \lg n)$$

pg 159 CLR

(7.4-1) Show that

$$T(n) = \max_{0 \le q \le n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n) \quad (*)$$

$\Rightarrow \quad T(n) = \Omega(n)$

$T(n) = \Omega(n^2) \Rightarrow \exists\, n_0 \ni \forall\, n \ge n_0 \quad T(n) \ge cn^2$.

The expression (*) is linear in n so shift the sequence until $n_0 \equiv 0$.

From I.E.

$$T(n-n_0) = \max_{0 \le q \le n-n_0-1} \left(T(q) + T(n-n_0-q-1)\right) + \Theta(n-n_0)$$

? is this correct?

what about the other limit?

$$\Rightarrow \quad T(\tilde{n}) = \max_{0 \le q \le \tilde{n}-1} \left(T(q) + T(\tilde{n}-q-1)\right) + \Theta(\tilde{n})$$

$\Rightarrow T(n) \overset{\text{guess}}{=} \max_{0 \le q \le n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n)$

$$T(n) \ge \max_{0 \le q \le n-1} \left(cq^2 + c(n-q-1)^2\right) + \Theta(n)$$

$$T(n) \ge c \max_{0 \le q \le n-1} \underbrace{\left(q^2 + (n-q-1)^2\right)}_{F(q)} + \Theta(n)$$

$F'(q) = 2q + 2(n-q-1)(-1)$

$= 2q - 2n + 2q + 2 = 4q - 2n + 2 = 0$

$q = \frac{2n-2}{4} = \frac{n-1}{2}$ This is a minimum

$F''(q) = 4 > 0$

$\therefore$

~~$T(n) \geq$~~ ~~$c\left(\frac{n-1}{2}\right)$~~

$F(0) = (n-1)^2$

$T(n) \geq c(n-1)^2 + \Theta(n)$ By excluding the maximum.

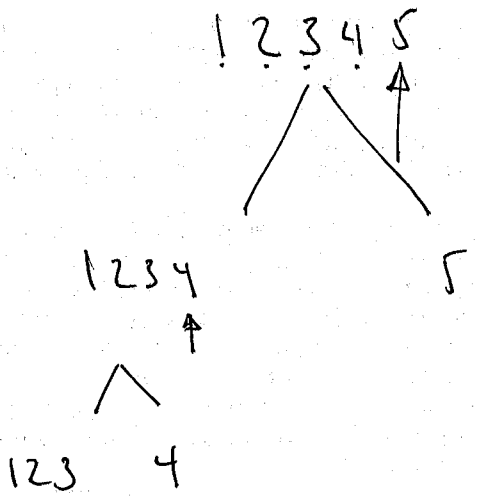
$\Rightarrow T(n) \geq cn^2 \quad \cdots \quad T(n) = \Omega(n^2)$

7.4.2 Quicksorts best case running time would be when no exchanges would ever have to be done in partition.

5 1 2 3 4
↑

What about an adaptive quicksort implementation? One that would see that he was exchanging a large # of elts & act accordingly.

Also one could take the mean of all the #'s & use that as a pivot for subarray. i.e. take the mean of each subarray we are working w/ & use that value as a pivot.



For this implementation of quicksort using the rightmost elt as the pivot would give the best case behavior when the input array is already sorted. & partition would require $\Theta(r-p+1)$ work.

~~Running this code would take~~

~~Partition~~ + Each partition would ~~require~~ be broken up into 2 pieces

of sizes 1 + n-1

$\therefore T(n) = \Theta(n) + T(1) + T(n-1)$

$\rightarrow T(n) = \Theta(n^2)$

⋮

Why is this not working ...?

$\Theta(n)$

$T(1) + \Theta(n-1)$

$T(1) + \Theta(n-2)$

⋮

$\sum_{k=} \Theta(k)$

7.4-3

$F(q) = q^2 + (n-q-1)^2$ achieves a maximum on $q = 0, 1, \ldots$

$F'(q) = 2q + 2(n-q-1)(-1)$

$= 2q - 2n + 2q + 2$

$F'(q) = 4q - 2n + 2$

$F''(q) = 4 > 0 \Rightarrow$ so if $F'(q)$ is a minimum

$F(0) = (n-1)^2$

$F(n-1) = (n-1)^2$

7.4-4 This can be shown by arguing that the best case running time of quicksort is $\Omega(n \lg n)$ randomizing cannot improve that.

7.4-5 For arrays of length k quicksort will simply return

Sort(A):

quicksort'(A)

for $i = 1$: $\lceil n/k \rceil$

call insertion sort $(A, i \cdot k, \min((i+1)k, n))$

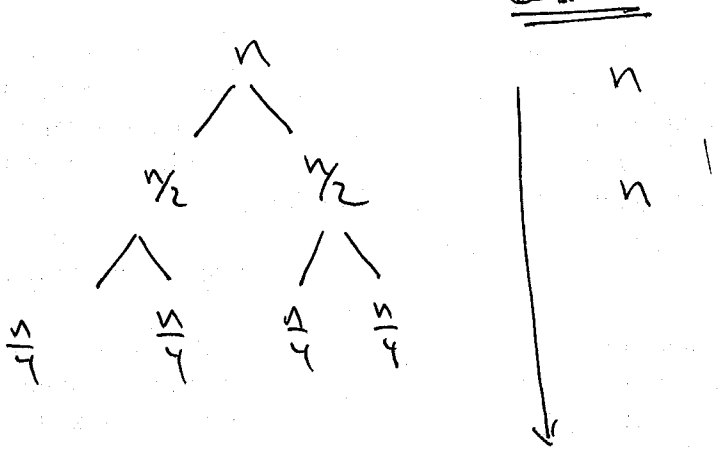
end for

$n = 10$ $\quad \frac{10}{3} = 3.\overline{3}$

$k = 3$

$\lceil \frac{n}{k} \rceil$ is the # of partitions to loop over

The call to quick sort paden for a random k



$$\left(\frac{n}{2^p}\right) = k$$

$$\Rightarrow n = k2^p$$

$$\lg\left(\frac{n}{k}\right) = p$$

Thus we have $p = \lg(n/k)$ divisions. W $\Theta(n)$ work at each level

$$= \sum_{i=1}^{\lg(n/k)} \Theta(n) = \underbrace{\Theta(n\lg(n/k))}_{\text{to perform the quicksort' portion}}$$

Then $\left\lceil \frac{n}{k} \right\rceil \cdot T_{\text{insertionsort}} = \left\lceil \frac{n}{k} \right\rceil \cdot k^2 = \Theta(nk)$

$T_{\text{insertion sort}}(n) = \Theta(n^2)$

Thus intotal I expect this portion to requir

$\Theta(n\lg(n/k)) + \Theta(n\lg(n/k) + nk)$

In theory pick k to minimize

$$\underbrace{O(n\lg(n/k) + nk)}_{F(k)}$$

$$F'(k) = (n\lg n - n\lg k + nk)'$$

$$= -\frac{n}{k} + n = 0$$

$\frac{n}{k} = n \Rightarrow k = 1$ ... this didn't seem to work so well...

In practice ... trial & error?

7.4-6 ($0 < \alpha < 1$)

By picking 3 elts at random & partitioning about their mean

each elt chosen is governed by a uniform probability distribution &

The mean will

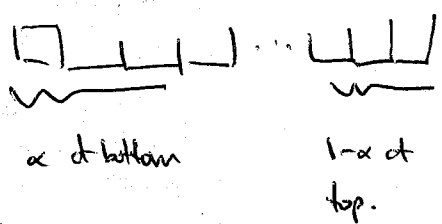
let $0 < \alpha < \frac{1}{2}$

$P_{\text{worse than } \alpha \text{ to } (1-\alpha) \text{ split}}$ = prob that the pivot elt returned from what ever strategy we use to select it is in the top (or bottom) range.

i.e., i"

= i.e. what is the probability that selecting the mean of 3 randomly chosen integers would yield either $\frac{1}{3}$ rd $1, 2, \ldots, \lfloor \alpha n \rfloor$

or elts $\lceil (1-\alpha)n \rceil, \ldots, n$



What is prob that randomly chosen elt will be in this range?

2α.

what is prob that 2nd randomly chosen elt will also be in this range

$(2\alpha)^2$

& 3rd $(2\alpha)^3$ This won't work for I will have elts in either

[redacted] end of the range

Ok a lower / upper bound on worst case could be gotten by assuming that you had to get all 3 random guesses to the the median from should be in the upper or lower extream.

$= x^3 + x^3 = 2x^3$ $\leq$ you'd get that ~~extream~~ mean value is found in one extream or the other.

∴ Worse split then x & $(1-x)$ is $< 2x^3$

7-1

(a) $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$ length 12.

Before loop. $p=1$, $r=12$

$x = 13$

$i = 0$

$j = 13$

do repeat $j \leftarrow j-1$ until $A[j] \leq x$ returns $j = 11$

do repeat $i \leftarrow i+1$ until $A[i] \geq x$ returns $i = 1$

exchange $A[i]$ + $A[j]$ gives

$A = \langle 6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21 \rangle$

---

do repeat $j \leftarrow j-1$ return $j = 10$

$i \leftarrow i+1$ " $i = 2$

exchanging gives

$A = \langle 6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21 \rangle$

$j = 9$

---

do repeat $j \leftarrow j-1$ until return $j = 9$

$i \leftarrow i+1$ until returns $i = 10$

Since $i \not< j$ return $j = 9$

(b) i starts at p-1 & is only modified on line 7.

line 7 is a addition of 1 to i.

The worry is that i may run off the end of the array A.

If I can show that $i < r \quad \forall i$

For $i \geq r$ i would

or 2) i pass j ie $i \geq j$

2 is not possible since ⇔ statement 9 would call Statement 11.

The same argument could be made for why j can never fall too low in the array.

(c) Since we know $p \leq j \leq r$ from the previous problem & the pseudocode we must show that $j \neq r$ when it returns & we are done then.

i

j=r

The only way j=r would be if it to equal $j \leftarrow j-1$ was

executed only once. This would mean that $A[r] \leq x$

w/ the 1st loop through the true while loop

$i = p$

& thus $A[p]$ & $A[r]$ would be exchanged.

Since the next loop through the while decreases j

$j \neq r$.

The basic idea is that one must loop at least once through the while true loop.

(d) (1) when an exchange takes place this requirement must be met. It is this exchange that guarantees this fact.

(e) Quicksort (A, p, r)

```
if p<r
  then q ← Hoare-partition(A,p,r)
       Quicksort(A, p, q-1)
       Quicksort(A, q+1, r)
endif
```

pg 160 CLR

08-06-02

7-2

(a) $X_i = I\{\text{ith smallest elt is chosen as the pivot}\}$.

$E[X_i] = \frac{1}{n}$

(b) $E[T(n)] = E\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q)) + \Theta(n)\right]$ (exp)

expected time of randomize quick sort = probability that each case happens * time for to sort those lists

(c)

$$= \Theta(n) + \frac{1}{n}\sum_{q=1}^{n} E[T(q-1)] + \frac{1}{n}\sum_{q=1}^{n} E[T(n-q)]$$

$k = n-q$

$q=1 \Rightarrow k = n-1$

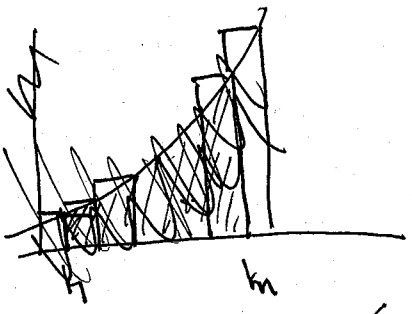
$q=n \Rightarrow k = 0$

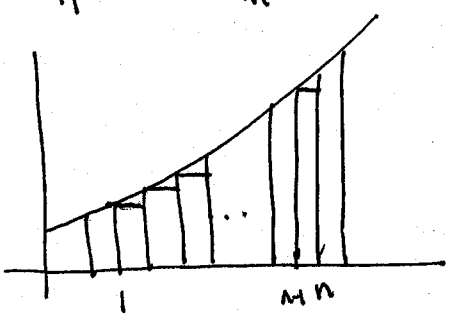
$$\frac{1}{n}\sum_{k=0}^{n-1} E[T(k)]$$

$$= \Theta(n) + \frac{2}{n}\sum_{q=0}^{n-1} E[T(q))]$$

(d) $k \lg k$ is monotonically increasing

$$\sum_{k=1}^{n-1} k \lg k \quad \text{is} \quad \int_{1}^{n} x \lg x \, dx$$





$$\int_1^n x \ln x \, dx = x(x\ln x - 1)\Big|_1^n -$$

$$\int v\,du = vu - \int u\,dv$$

$$= \frac{x^2}{2}\ln x\Big|_1^n - \int_1^n \frac{x^2}{2}\cdot\frac{1}{x}\,dx$$

$$\int x \ln x \, dx = \frac{x^2}{2}\ln x - \frac{1}{2}\int x\,dx$$

$$= \frac{x^2}{2}\ln x - \frac{x^2}{4}$$

$$\int_1^n x\ln x\,dx = \frac{n^2}{2}\ln n - 0 - \frac{n^2}{4} + \frac{1}{4}$$

Following hint

$$\sum_{k=1}^{n-1} k\lg k = \sum_{k=1}^{\lceil n/2\rceil - 1} k\lg k + \sum_{k=\lceil \frac{n}{2}\rceil}^{n-1} k\lg k \le 2\sum_{k=\lceil \frac{n}{2}\rceil}^{n-1} k\lg k$$

$$\leq 2\int_{\lceil \frac{n}{2} \rceil}^{n} x \lg x \, dx \qquad \lg x = \frac{\ln x}{\ln 2}$$

$$= \frac{2}{\ln 2}\int_{\lceil \frac{n}{2} \rceil}^{n} x \ln x \, dx = \left(\frac{2}{\ln 2}\right)\left[\frac{x^2}{2}\ln x - \frac{x^2}{4} \right|_{\lceil \frac{n}{2} \rceil}^{n}$$

$$= 2\left[\frac{x^2}{2}\lg x - \frac{x^2}{4\ln 2} \right|_{\lceil \frac{n}{2} \rceil}^{n}$$

$$= 2\left[\frac{n^2}{2}\lg n - \underbrace{\frac{n^2}{4\ln 2}} - \frac{n^2}{8}\lg n + \underbrace{\frac{n^2}{4\ln 2 \cdot 4}} \right]$$

$$= 2\left[\frac{n^2}{2}\lg n + \frac{n^2}{4\ln 2}\underbrace{\left[-1+\frac{1}{4}\right]}_{-\frac{3}{4}} - \frac{n^2}{8}\lg n \right]$$

?? It no 2

where does 2 go? can't be correct?

$$\leq \frac{n^2}{2}\lg n - \frac{n^2}{8}$$

(e) Pf $E[T(n)] \equiv E(n) = \Theta(n \lg n)$

Ass $E[T(n)] \le an\log n - bn \quad \forall\ n \le n_0$

Th

$$E[T(n)] \le \frac{2}{n}\sum_{q=0}^{n-1}(aq\log q - bq) + \Theta(n)$$

$$= \frac{2}{n}a\sum_{q=0}^{n-1} q\log q - \frac{2b}{n}\sum_{q=0}^{n-1} q + \Theta(n)$$

$$\le \frac{2}{n}a\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) - \frac{2b}{8n}\Theta(n^2) + \Theta(n)$$

$$= an\log n - \frac{2}{8}an - 2b\Theta(n) + \Theta(n)$$

$\Theta(n)$

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

7-3

(a) This can be shown if $i-j+1=2$ works ✓

Then Assume length 2 array

Stooge-sort works for an array of length k.

$i+1-j \geq 0$

$-i-1+j \leq 0$

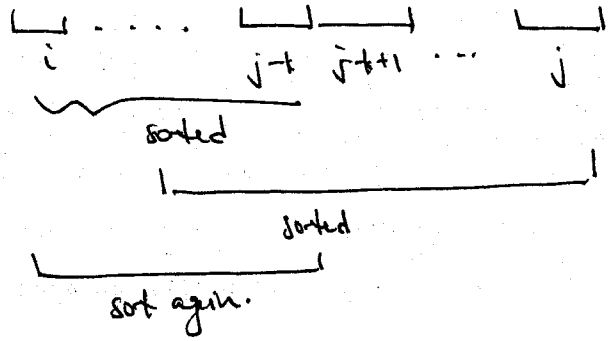
$j-i-1 \leq 0$

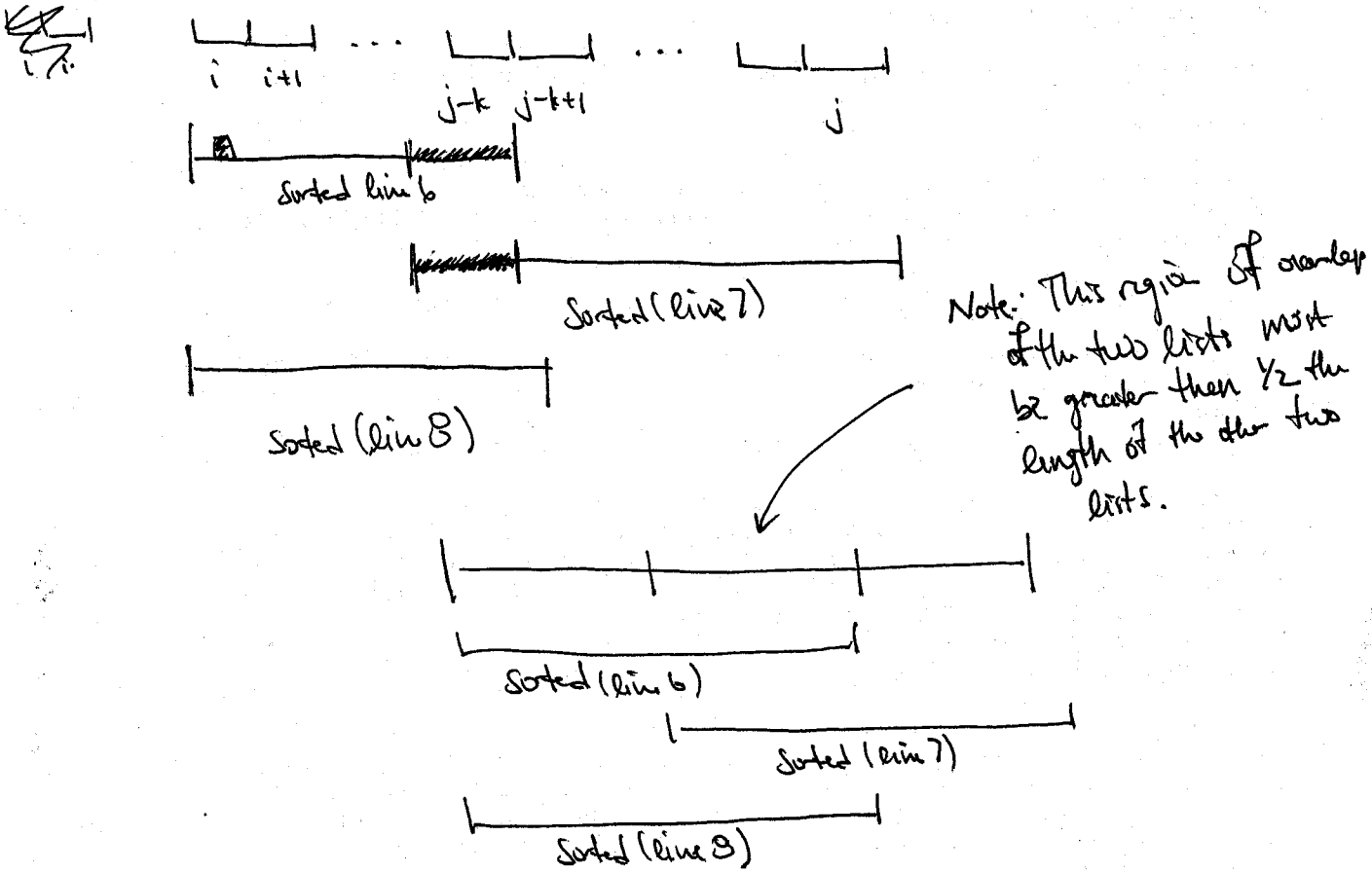
$j-i+1 \leq 2$

Pf: That it works for an array of length $k+1$

Since Stooge-sort$(A,i,j-t)$ returns correctly Stooge-sort$(A,\cdot$.

↓ The 3rd call to Stooge-sort$(A,i,j-t)$ moves any required elts into the 1st (bottom) half.





(b)

$T(n) = \Theta(1) + 3T(n/3)$ ← error... each list is $\frac{2}{3}n$ long

$= \Theta(1) + 3(\Theta(1) + 3T(n/9))$

$= \Theta(1) + \Theta(3) + 9T(n/9)$

$= \sum_{k=0}^{p-1} \Theta(3^k) + 3^p T(n/3^p)$

Stop when $\frac{n}{3^p} \leq 1$

$\Rightarrow n = 3^p$

$p = \log_3 n$

$\Rightarrow T(n) = \Theta\left(\sum_{k=0}^{\log_3 n - 1} 3^k\right) + 3^{\log_3 n} T\left(\frac{n}{3^{\log_3 n}}\right)$

$= n + \Theta\left(\frac{3^k}{3-1}\Big|_0^{\log_3 n}\right)$

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

$T(n) = \Theta(1) + 3T(\frac{2}{3}n)$

$= \Theta(1) + 3(\Theta(1) + 3T((\frac{2}{3})^2 n))$

$= \Theta(1) + \Theta(3) + 3^2 T((\frac{2}{3})^2 n)$

$= \sum_{k=0}^{p-1} \Theta(3^k) + 3^p T((\frac{2}{3})^p n)$

Stop when $(\frac{2}{3})^p n = 1$

$\Rightarrow n = (\frac{3}{2})^p$

$p = \frac{\log n}{\log(3/2)}$; $\frac{\log_3 n}{\log_3(3/2)} = p$

$$T(n) = \Theta\left(\frac{3^k}{3-1}\Big|_0^p\right) + 3^p T(1)$$

$$= \frac{1}{2}\Theta(3^p) + 3^p T(1)$$

$$= \Theta(3^p) = \Theta(3^{\frac{\log_3 n}{\log_3(3/2)}}) = \Theta(\left[3^{\log_3 n}\right]^{\frac{1}{\log_3(3/2)}})$$

$$= \Theta(n^{\frac{1}{\log_3(3/2)}}) = \Theta(n^{\frac{\ln(3/2)}{\ln 3}})$$

$$\log_3(3/2) = \frac{\ln(3/2)}{\ln 3}$$

(c) Worst case running time for

insertion $\Omega(n^2)$

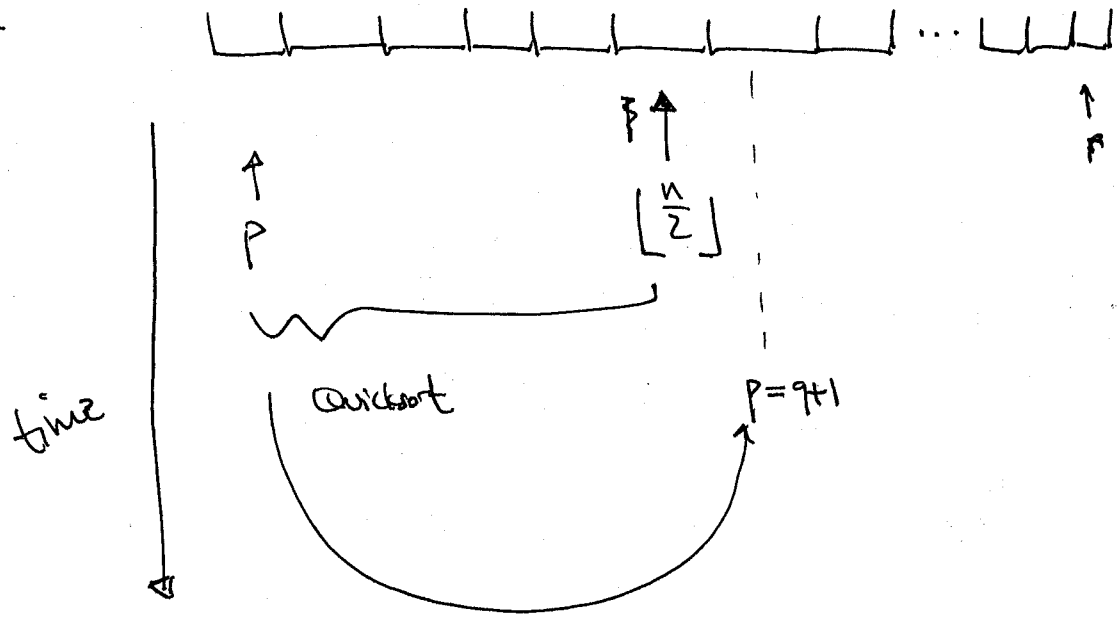
merge sort $\Omega(n \lg n)$

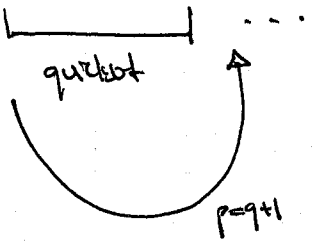
heap sort $\Omega(n \lg n)$

quicksort $\Omega(n^2)$

7-4

(a)





this routine simply does what quicksort would do but it does not call quicksort recursively on the 2nd (right) subarray instead the left most pointer is moved to

This will sort an array A as q (latter portion) is the index to all elts. to the left of q or less than all elts. to the right of q.

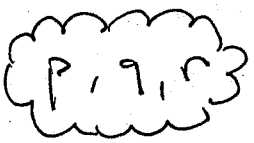
(b) For a stack depth of $\Theta(n)$ we must make $\Theta(n)$ recursive calls.

If A is the constant array q from partition maybe equal to n & quicksort will be called w/ arguments (A,1,n-1)

(c) while p<r

do

q ← partition(A,p,r)

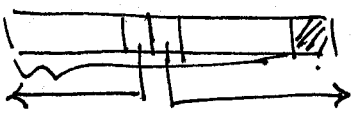


if (p-q+1 < r-q+1) then /* left partition is smaller */

Quicksort'(A, p, q)

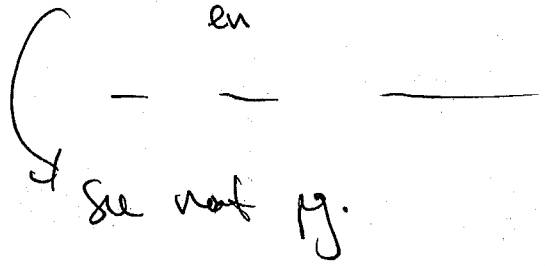
else

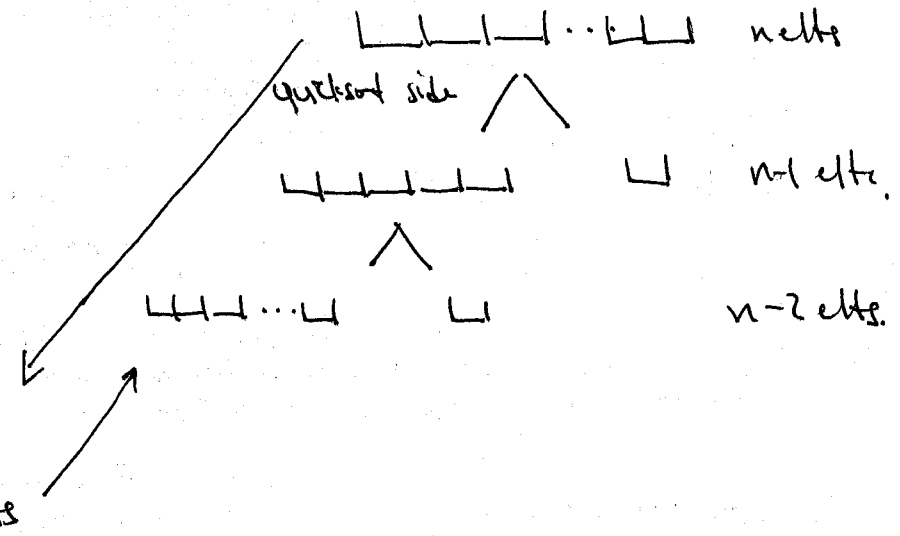
Quicksort'(

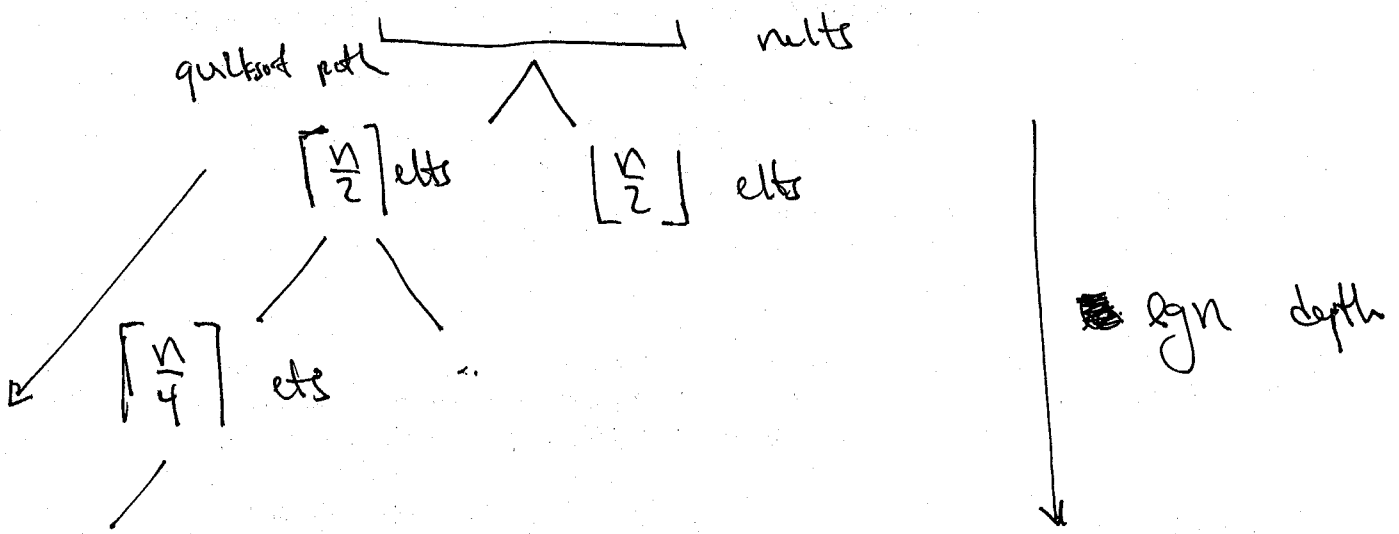


en

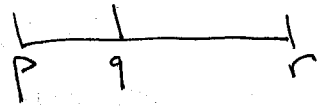
— — ——







```
while p < r
  do
    q ← partition(A, p, r)
    if (p-q+1 < r-q+1) then
       quicksort'(A, p, q)
        p ← q+1
    else
       quicksort'(A, q+1, r)
        r ← q
    endif
  enddo.
```



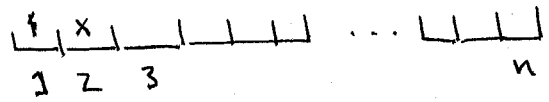
/* left partition is smaller */

/* right partition is ~~longer~~ smaller */

The arguments given on pg 157 show that the partitioning does not have to be as good as $\frac{n}{2}$ at each level to still give $O(\lg n)$ depth.

7-5

(a) $P_1 = 0 = P_n$

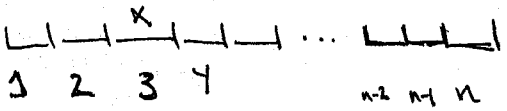


$P_2 = \left(\frac{1}{n}\right)\cdot$ ← 2nd elt picked can end up in any spot from

↑ 1st elt picked must end up in location #1

$$P_2 = \left(\frac{1}{n}\right)\left(\frac{1}{n}\right)\left(\frac{n-3+1}{n}\right) = \frac{n-2}{n^3}$$

↑ 2nd elt picked must end up in spot #2



$$P_3 = \left(\frac{3-1}{n}\right)\cdot\frac{1}{n}\cdot\left(\frac{n-4+1}{n}\right) = \frac{2(n-3)}{n^3}$$

↑ 1st elt can be in either spot 1 or spot 2

⋮

$$P_i = \left(\frac{i-1}{n}\right)\left(\frac{1}{n}\right)\left(\frac{n-(i+1)+1}{n}\right) = \left(\frac{i-1}{n}\right)\left(\frac{1}{n}\right)\left(\frac{n-i}{n}\right) = \frac{(i-1)(n-i)}{n^3}$$

Check $P_1 = 0$ ✓ $P_2 = \frac{n-2}{n^3}$ ✓ $P_3 = \frac{2(n-3)}{n^3}$ ✓ ... $P_n = 0$.

Check $\sum_{i=1}^{n} P_i \overset{?}{=} 1$

$$\sum_{i=1}^{n} \frac{(i-1)(n-i)}{n^3} = \frac{1}{n^3}\sum_{i=1}^{n}(i-1)(n-i)$$

$$= \frac{1}{n^3}\Big[\ 0 + 2(n-1)$$

$$= \frac{1}{n^3}\underbrace{\sum_{i=1}^{n}(ni - i^2 - n + i)}_{i=1} = \frac{1}{n^2}\sum_{i=1}^{n} i - \frac{1}{n^3}\sum_{i=1}^{n} i^2 - \frac{1}{n^2}\sum_{i=1}^{n} 1 + \frac{1}{n^3}\sum_{i=1}^{n} i$$

$$= \frac{1}{n^2}\,\frac{n(n+1)}{2} - \frac{1}{n^3}$$

(b) ordinary impl $P_{x^*} = \frac{1}{n}$

median of 3 implementation $= P_{x^*} = \dfrac{\left(\left\lfloor\frac{n+1}{2}\right\rfloor - 1\right)\left(n - \left\lfloor\frac{n+1}{2}\right\rfloor\right)}{n^3}$

$$\frac{\left(\frac{n+1}{2} - 1\right)\left(n - \frac{n+1}{2}\right)}{n^3} =$$

$$= \frac{\left(\frac{n-1}{2}\right)\left(\frac{2n-n-1}{2n}\right)}{n^3} = \frac{\left(\frac{n-1}{2}\right)\left(\frac{n-1}{2n}\right)}{n^3} = \left(1 - \frac{1}{n}\right)\left(\frac{1}{2} -$$

$$= \frac{\left(\frac{n+1}{2} - \frac{2}{2}\right)\left(\frac{2n}{2} - \left(\frac{n+1}{2}\right)\right)}{n^3} = \frac{\left(\frac{n-1}{2}\right)\left(\frac{n-1}{2}\right)}{n^3}$$

$$= \frac{1}{4n}\left(1 - \frac{1}{n}\right)^2 \Rightarrow \frac{1}{4n} \qquad n \gg 1$$

Why is this probability smaller than that desired above? It should be greater. Actually, I think this is correct. It would seem harder to pick all 3 elts just correctly so that the true median of the list was returned.

(c) $\text{prob}(\underset{\text{method}}{\text{good split original}}) = \dfrac{\frac{2n}{3} - \frac{n}{3} + 1}{n} = \dfrac{\frac{n}{3}+1}{n} = \frac{1}{3} + \frac{1}{n}$

$$\text{prob}(\underset{\text{method}}{\text{good split new}}) = \sum_{i=\frac{n}{3}}^{\frac{2n}{3}} p_i$$

$$= \sum_{i=\frac{n}{3}}^{\frac{2n}{3}} \frac{(i-1)(n-i)}{n^3} = \frac{1}{n^3} \sum_{i=\frac{n}{3}}^{\frac{2n}{3}} (i-1)(n-i) = \frac{1}{n^3}$$

(d) At each level the work is dominated by the partition routine. Adding a median calculation will not affect anything from this except to add an $O(\lg n)$ constant factor.

7-6

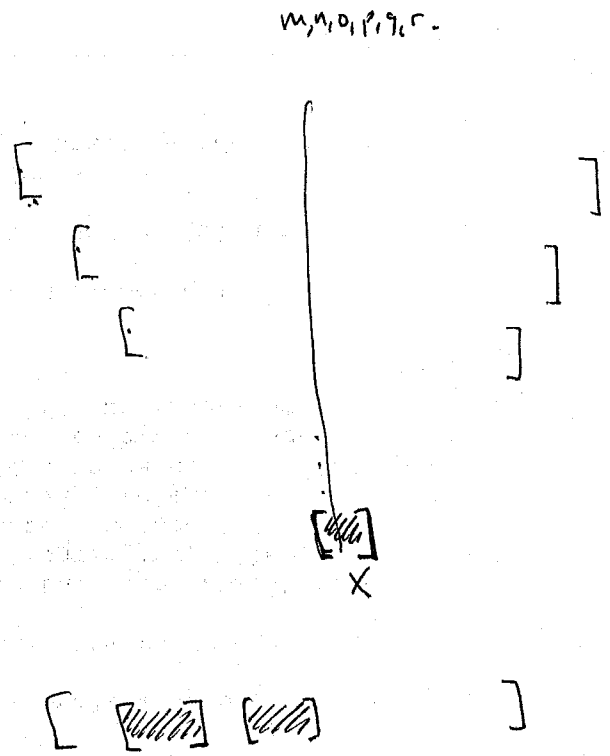
(a) $[a_{i_1}, b_{i_1}]$
$[a_{i_2}, b_{i_2}]$
$[a_{i_3}, b_{i_3}]$
...

quicksort($=[a_i], \cdots$)

$A = [a_1, a_2, \ldots, a_n]$
$B = [b_1, b_2, \ldots, b_n]$

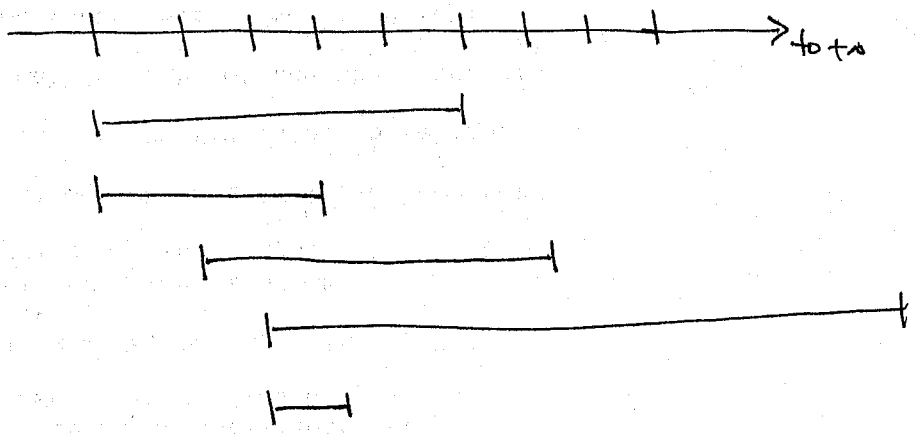
fuzzy-sort(A, B, p, r)
if p < r then
q ←

m, n, o, p, q, r.



Note: If all the intervals overlap then a common elt for c_i will work for each interval

$c_1 \leq c_2 \leq \cdots \leq c_n$



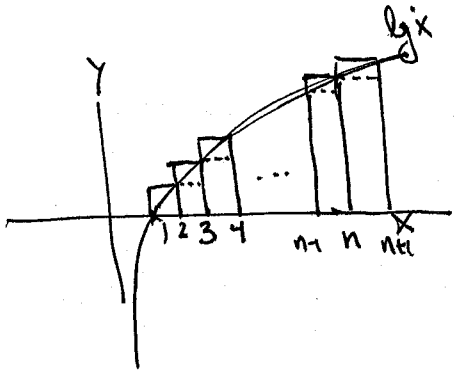
Not sure how to solve this problem...

8.1-1 $n-1$ if all ~~nots~~ elts are in order, then ~~sorting~~ performing comparisons would only need to be done $n-1$ times.

8.1-2

$$\lg(n!) = \lg(n(n-1)\cdots 2\cdot 1)$$

$$= \sum_{k=1}^{n} \lg(k)$$



$\lg(k)$ is a monotonically increasing fn:

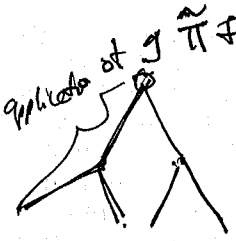
$$\int_{0}^{n} \lg x\,dx \le \sum_{k=1}^{n} \lg(k) \le \int_{1}^{n+1} \lg(x)\,dx$$

$$\Downarrow$$

$$\int_{1}^{n} \lg x\,dx \le \sum_{k=2}^{n} \lg(k) \le \int_{2}^{n+1} \lg(k)\,dk$$

8.1.3 Assume that there existed a comparison sorting method that uses linear on at least ½ of the $n!$ possible inputs.

Then the decision tree that represented the sorting process would have to have at least $\frac{n!}{2}$ linear legs.



This problem is equivalent to what fraction of a full binary tree has path from the root to the leaf that is linear?

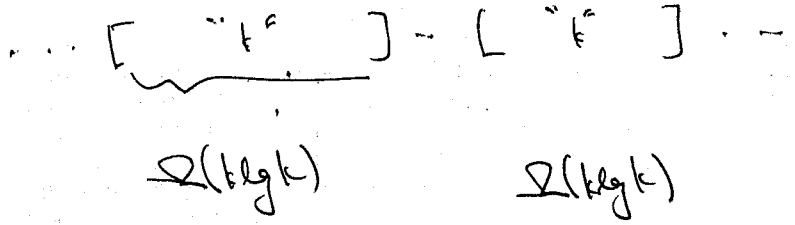
$\tilde{\pi}$ is allowed to only change 2 consecutive elts. one at a time.

Not so?

Pg 168 CLR

DB-DB-02-1

8.1-4



$\Omega(k \lg k)$ $\Omega(k \lg k)$

$$\left(\frac{n}{k}\right) \Omega(k \lg k) = \Omega(n \lg k)$$

? Not sure ?

A = 2 5 3 0 2 3 0 3

C = 2 2 4 7 7 8

$j = n$:

$B[C[A[n]]] = B[C[3]] = B[7] = 3$

$j = n-1$

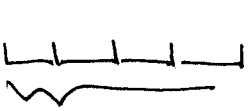
$B[C[0]] = B[2] = 0$

$j = n-2$

$B[C[3]] = 3$

$\parallel$

elts ≤ 3

Boxy: 

next C[3]
spaces "below"
I can place 3 in the Boxy

Pg 170 CLR

08-14-021

8.2-1

$\langle 6,0,2,0,1,3,4,6,1,3,2 \rangle$

6 0 2 0 1 3 4 6 1 3 2 $\quad k=6$

C: 2 2 2 2 1 0 2 { C[i] counts the # of elts w/ value i. in array A

index: 0 1 2 3 4 5 6

C: 2 4 6 8 9 9 11 { C[i] now contains the # of elts less than or equal to i

index: 0 1 2 3 4 5 6

B: 0 0 1 1 2 2 3 3 4 6 6

index: 1 2 3 4 5 6 7 8 9 10 11

8.2-2 Assum $A[i] = A[j]$ w/ $i < j$

The stability follows from the fact that we place the left most elts in array A in the left most positions. The example above shows the stability.

8.2-3

The algo still works but is no longer ~~stable~~ stable

8.2-4

Perform 4 to 6 including step 7 of a counting sort.

Then to answer the given query compute

$C[b]$ = # elts in array A less than or equal to b

$C[a]$ = " " " " " " " " " " " " a

$C[b] - C[a]$ = # elts in array A between $\beta[a, b]$

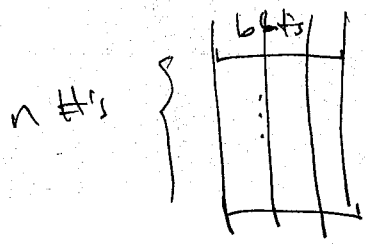
Thus missing # elts equal to a.

$\therefore C[a] - C[a-1]$

$\therefore$ $C[b] - C[a-1]$ = # elts in array A between a & b inclusive.

pg 172 CLR

08-02-14-02 1



$\Theta(n+k) = \Theta(n+2^r)$

$F(r) = \left(\frac{b}{r}\right)(n+2^r)$ $\quad b < \lfloor \lg n \rfloor$

$b \geq \lfloor \lg n \rfloor$ $\quad$ pick $r = \lfloor \lg n \rfloor$

$$\frac{b}{\lfloor \lg n \rfloor}\left(n + 2^{\lfloor \lg n \rfloor}\right) = \Theta\left(\frac{bn}{\lg n}\right)$$

Pg 173 CLR

08-14-02 1

Q3-1

COW 12
DOG 5
SEA 1
RUG 6
ROW 13
MOB 3
BOX 15
TAB 4 ⇒
BAR 9
EAR 10
TAR 11
DIG 7
BIG 8
TEA 2
NOW 14
FOX 16

SEA 5
TEA 6
MOB 9
TAB 1
DOG 10
RUG 16
DIG 7
BIG 8
BAR 2
EAR 3 ⇒
TAR 4
COW 11
ROW 12
NOW 13
BOX 14
FOX 15

TAB 14
BAR 1
EAR 7
TAR 15
SEA 13
TEA 16
DIG 5
BIG 2
MOB 9
DOG 6
COW 4
ROW 11
NOW 10
BOX 3
FOX 8
RUG 12

BAR
BIG
BOX
COW
⇒
DIG
DOG
EAR
FOX
MOB
NOW
ROW
RUG
SEA
TAB
TAR
TEA

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

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

$$= \Theta(n) + \Theta\left((2 - \tfrac{1}{n})(n)\right) = \Theta(n)$$

Pg 173 CLR

8.3-2

Stable (Yes/No)

insertion sort — depends on implementation but can be made so

merge sort — depends on implement " " " " "

? heap sort — Can be made so, by insuring that " Building the "

quick sort — yes.

Can I Check answer afterwards? Don't see how

Can I modify the input list. form ~~C[i] as in counting sort where~~

~~C[i] = #elts~~ I could modify the elts that are the same value so that when sorted they retain the original order,

1) means the smallest + next smallest elt in the array.

Sthen takes O(n).

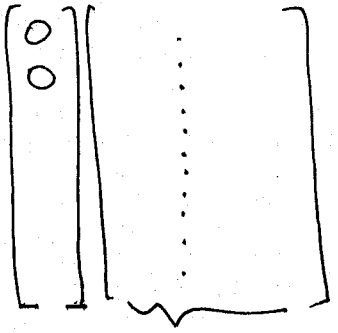
8.3-3 Induction $d = 1$, a stable sort sorts these elts.

Assum radix sort works for $d = k$.

⇒ given n k digit integers radix sort correctly sorts them.

Assum we have n $k+1$ digit integers apply radix sort to the last k digits of our set of #'s

Thus gives us a set of n #'s



k digits correctly sorted.

Now sort the 1st ~~digit~~ column on its digit since we are using a stable sort for this column the result will be correct.

8.3-4 $n^2 - 1 = (n+1)(n-1)$

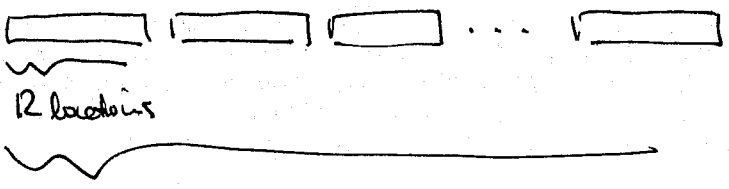
use radix sort w/ d = ?

k = 9

n given

?

8.3-5

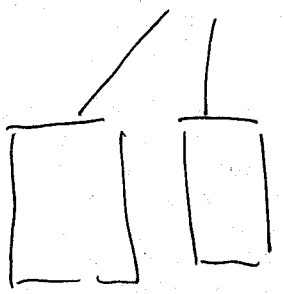


80 columns in general d columns

Cards (column oriented) ⇒ ~~10 bins for #'s~~

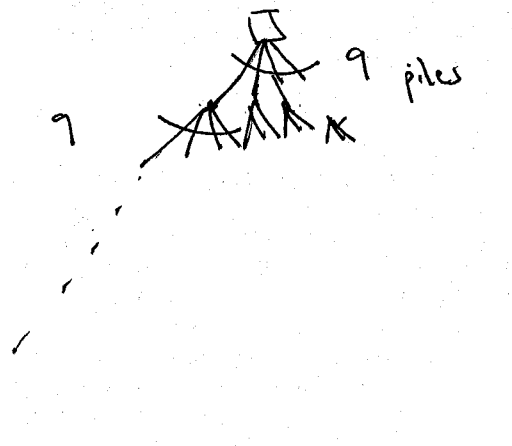
Here I will sort from max to min or descending sort

For the 1st pass all cards would be seperated into 12 piles where the 1st pile is say the largest 1st digit or all legal #'s appear in this pile



one would then have to set 9 piles aside, set the machine to sort based on the 2nd column location.

Again setting 9 piles aside



sorting passes =

nodes in a k-ary tree w/ k = 9 + depth d

d >> k

depth of d

$\sim O(9 \cdot d)$

piles = $O(9 \cdot d)$

Pg 177 CLR

8.4-1

$A = \langle . \ . \ . \ . \rangle$ $n = 10$ 1st place elts into buckets.

A	B
1	0
2	1 .13 .16
3	2 .2
4	3 .39
5	4 .42
6	5 .53
7	6 .64
8	7 .79 .71
9	8 .89
10	9

Now sort the buckets w/ insertion sort say...

.13 .16

.2

.39

.42

.53

.64

.71 .79

.89

8.4-2 If all elts get mapped to the same bucket we are w/ a rate of insertion sort of $\Theta(n^2)$

Change insertion sort to an sort that runs w $O(n \lg n)$ like heapsort or quicksort or merge

8.4-3 ~~$E[X] = \frac{1}{2}$~~ HH, HT, TH, TT

X: 2 1 1 0

$$E[X] = p_1 \cdot X_1 + p_2 X_2 + p_3 X_3 + p_4 X_4$$

$$= \frac{1}{4}(2+1+1) = \frac{4}{4} = 1$$

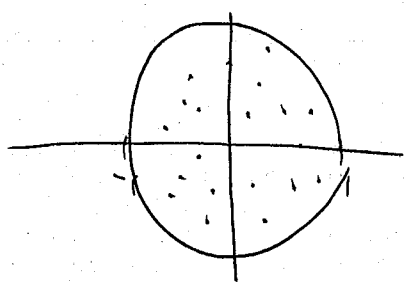
$$E[X^2] = p_1 X_1^2 + p_2 X_2^2 + p_3 X_3^2 + p_4 X_4^2$$

$$= \frac{1}{4}(4+1+1) = \frac{6}{4} = \frac{3}{2}$$

~~$E[(X - \bar{X})^2]$~~

$E[X]^2 = 1$

8.4-4



$\forall$ point compute d_i (req $O(n)$ work)

Then breaking the ~~radius~~ radius up into say 100 segments perform bucket sort on

$\{d_i\}$

uniform distribution of the distance from 0 to 1.

Break up into n intervals

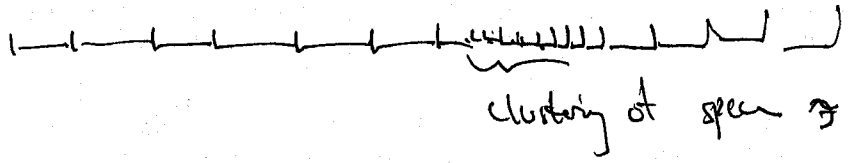
results follows from bucket sort

8.4-5 $X_1, \ldots, X_n$ Not for uniform ~~scaling~~ P.D.F's

the spacing of the interval between 2 elts is uniform,

The idea is you want a constant # of elts of your array in each bucket, Not a huge # of elts in 1 bucket or the other.

Thus one would have to make the intervals smaller closer to the elts w/ greater probability of appearance



uniform PDF $\Delta x_i \sim \frac{1}{n}$

nonuniform PDF $\Delta x_i \sim$

elts we expect to find in interval $0, \Delta x$ is

$$P\{X < \Delta x\} = \frac{1}{n_{\Delta x}}$$

" " " $\Delta x, 2\Delta x$

$$P\{X < 2\Delta x\} - P\{X < \Delta x\} = \frac{1}{n_{\Delta x}} \qquad n_{\Delta x} = \#\text{ of } \Delta x \text{ intervals}$$

$\therefore$

~~diff $\Delta x_1 \Rightarrow P\{X < \Delta x_1\} = \frac{1}{n}$~~ $n = \#$ of elts to be sorted

" $\Delta x_2 \Rightarrow$ ~~$P\{X < \Delta x_1 + \Delta x_2\} - P\{X$~~

$P\{\Delta x_1 < X < \Delta x_2$

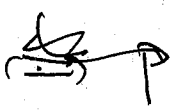
Define the partition of the range of the #'s as follows:

Assume range of #'s = $[0,1]$

define $x_0 = 0$

define x_1 : $P\{\overline{X} < x_1\} = \frac{1}{n}$ n = # of elts to be sorted

define x_2 : $P\{x_1 < \overline{X} < x_2\} = \frac{1}{n}$



$\vdots$

x_n : $P\{x_n < \overline{X} < 1\} = \frac{1}{n}$

If P as in book is defined as

$P(x) = P\{\overline{X} < x\}$ then

$P\{x_1 < \overline{X} < x_2\} = P(x_2) - P(x_1) = \frac{1}{n}$

$P(x_2) = \cdots$

Solve for x_2.

PJ 17.8 CLR 08-15-02

(a) Prob of an input sequence having any specified permutation is $\frac{1}{n!}$

of which there are $n!$ that. The rest are not probable & have probability 0.

(b) $D(T) = \sum_{l \in \text{leafs}} \text{depths}(l) = k +$?

Don't follow

(c) ?

(d) .. Assn $i = \frac{k}{2}$

$$d(k) = \min_{1 \le i \le k-1} \{ d(i) + d(k-i) + k \}$$

If $d(i) = i \lg i$

$$d(k) = \min_{1 \le i \le k-1} \{ i \lg i + (k-i) \lg (k-i) + k \}$$

$$= \frac{k}{2} \lg\left(\frac{k}{2}\right) + \frac{k}{2} \lg\left(\frac{k}{2}\right) + k$$

$$= k \lg k - \lg 2 + k = k \lg k = \Omega(k \lg k)$$

(c) Since $k = n!$

$D(T_A) = \Omega(n! \lg(n!))$

~~$\ln(n!) = n \ln n - n$~~

~~$\frac{1}{n!}$~~ ~~$\frac{1}{n!}$~~

Then expected Time

$$T(n) = \sum_i p_i D(l_i) = \sum_i \left(\frac{1}{n!}\right) D(l_i)$$

$$= \frac{1}{n!} \sum_i D(l_i) = \frac{1}{n!} D(T_A) = \Omega(\lg n!)$$

$= \Omega(n \lg n)$ By stirling's formula.

(I) ?

Pg 178 CLR 08-14-02 /

8.2

(a) Counting sort satisfies both these properties

(b) Counting sort wont work since it requires auxillary arrays.

Using two pointers, one at the beginning + 1 at the end of the list we move our pointers towards the center exchanging any 2 elts when the left most elt is a 1 + the right most elt is a 0.

(c) Many of the sorting routines can be written stably. insertion sort for instance, giving the fastest quicksort will work

(d) Algo in pt (a) will work since
(b) will not work since subsorting routin is not stable.
(c) will ~~too~~ not meet the time requirements of O(bn)

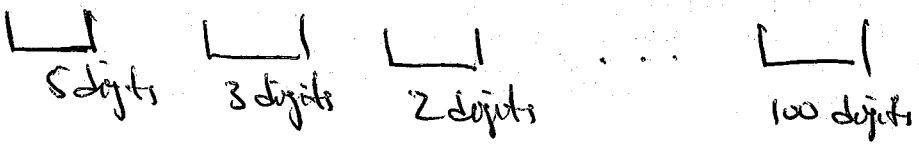
(e) Don't understand why I have to modify anything algo. works as is. If you don't like the fact that the range of the elts is A is 1 to k rather than 0 to k simply subtract one from ~~the~~ each elt of A.

pg 179 CLR

8-16-02 1

B-3

(a)

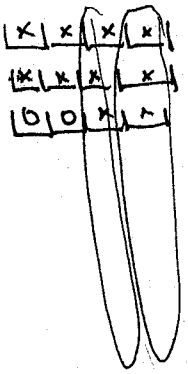


Assume no integers are prepended w/ zeros.

Then the #'s w/ larger # of digits should be sorted last.

Use Bucket sort w/ each bucket corresponding to the # of digits used

# of digits	B
1	
2	
3	
⋮	
n	



1) Find the longest integer (# of characters) and pad all other integers w/ zeros.

2) Run Radix sort on this list requires

$O(d \cdot n)$ w/ d = # of columns & n = length of each column

Using radix sort w/ a notion that we are to line all #'s up against the same location & if a # does not have a digit in the column being searched on it will not be placed at the bottom i.e. a "blank" is a character less than 1.

(b)

1) Line all elts up w/ the left border.
2) Do stable sort based on 1st elt of word
3) Do stable sort based on 2nd column when a blank is

a —
b — x
a —

won't work ...

Maybe here use bucket sort.

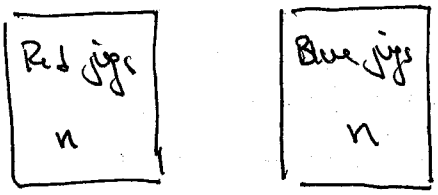
a] a ab ...

b]

Not sure.

Pg 179 CLR

(B-4)



] matching of every red jug onto every Blue jug.

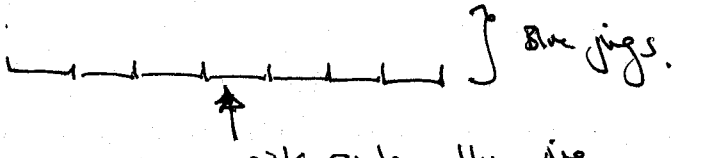
Assume that the red jugs have been labeled 1:n
& sim for the blue jugs.

(a) By picking a Red jug & comparing it w every Blue jug until a match is found. Can has an $O(n^2)$ algo

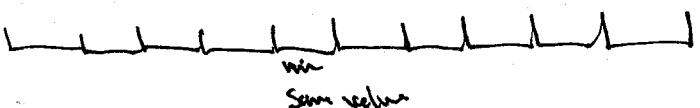
(b) Because any algo with that solves this problem must be able to determine (using binary decisions only) one of the $n!$ permutations of the integers from 1 to n
A decision tree argument shows a lower bound of

$\Omega(n \lg n)$

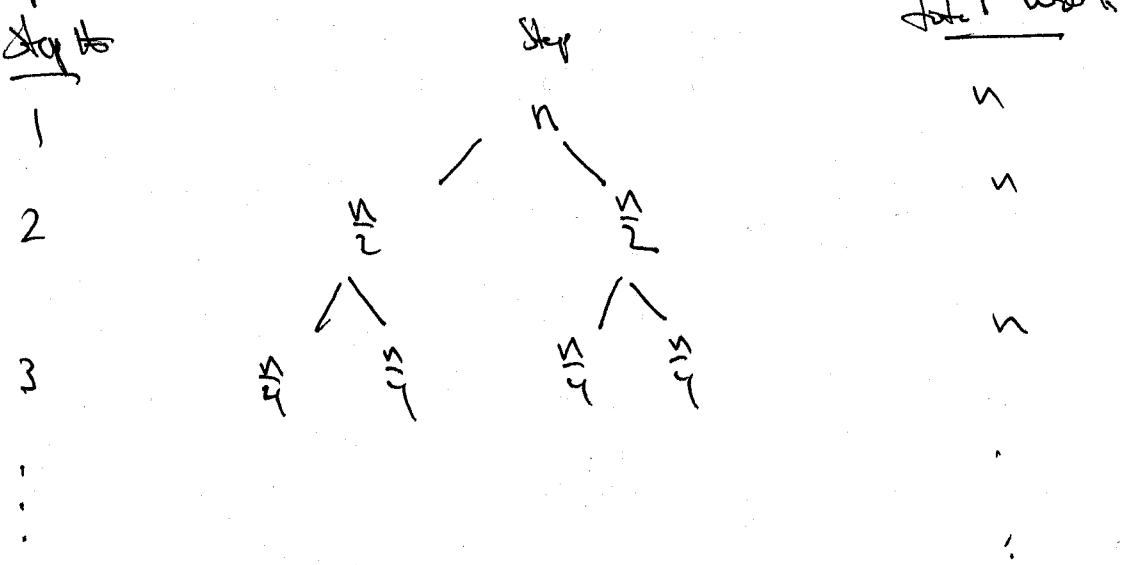
(c)



partition the red jugs based on this jug



The ~~&~~ Using the equivalent jig ~~in~~ in the red jigs partition the blue jigs.



Now to 2 subproblems when you pick a jig at random from the left half & partition both red & blue arrays about that value. We move roughly ~~&~~ n/2 elts but have to do this twice (once for each partition)

We hope that each partition is split by $\frac{n}{2}, \frac{n}{2}$

Worst case it will be split 1 & n-1.

Then work is $O(n^2)$

This is analogous to the quick sort procedure

8-5

(a) 1-sorted ⇒ sorted in the regular sense

(b) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Think: 2, 1, 4, 3, 6, 5, 8, 7, 10, 9

Check 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5

(c) Pr: n-elt array is k-sorted if $A[i] \le A[i+k]$ $\forall\ i = 1, 2, \ldots, n-k$.

Assume not. Then $A[i] > A[i+k]$ for $i = i^* \in [1, n-k]$

Then $$\frac{1}{k}\sum_{j=i}^{i+k-1} A[j] = \frac{A[i]}{k} + \frac{1}{k}\sum_{j=i+1}^{i+k-1} A[j]$$

$$> \frac{A[i+k]}{k} + \frac{1}{k}\sum_{j=i+1}^{i+k-1} A[j] = \frac{1}{k}\sum_{j=i+1}^{i+k} A[j] \quad \rightarrow\!\!\!\leftarrow$$

to definition of k-sorted.

Pr: if $A[i] \le A[i+k]$ $\forall\ i = 1, 2, \ldots, n-k$ then A is k-sorted

$$\frac{1}{k}\sum_{j=i}^{i+k-1} A[j] \le$$

$$\frac{1}{k}A[i] + \frac{1}{k}\sum_{j=i+1}^{i+k-1} A[j] \le \frac{A[i+k]}{k} + \frac{1}{k}\sum \le \sum_{j=i+1}^{i+k} A[j]$$

(d) We are required to find an algorithm that will return array A s.t. $A[i] \leq A[i+k] \quad \forall\ i=1,2,3,\ldots,n-k$

?

(e) Ex 6.5-8 gives an algorithm to merge k sorted lists into 1 sorted list in $O(n \lg k)$

? The question is because I don't see that the k lists I am left with are actually sorted. just average sorted

(f) know that $\underline{O(n \lg (n/k))} = \Omega(n \lg (n/k))$ By lower bound on comparison sorts

?

pg 180 CLR

B-6

(a) if/ the list is sorted already, then separating them into 2 lists would be picking n elts from the input $2n$

$$\binom{2n}{n}$$

(b) By adj

$$\binom{2n}{n} \leq l \leq 2^h$$

$$\binom{2n}{n} = \frac{2n!}{n!(n!)} = \frac{2n(2n-1)(2n-2)\cdots(n+1)}{n!}$$

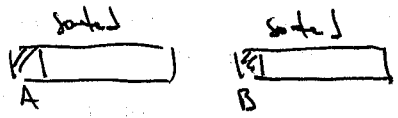
$$h \geq \lg\binom{2n}{n} = \lg(2n!) - \lg(n!)^2$$

$$\cong (2n)\lg(2n) - (\not{2n}) - 2[n\lg n - \not{n}]$$

$$= 2n\lg(2n) - 2n\lg n$$

$$= 2n\lg(2) + 2n\lg n - 2n\lg n = 2n + o(n)$$

(c)



AB : if A & B were not compared one would not know which elt came 1st.

(d) Consider the worst case situation of elts alternating every other one.

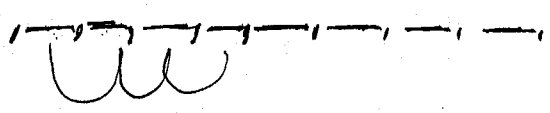
13579		2468

Then $2n-1$ comparisons need to be made $2n-1$ ← don't need to compare the last elt, since there is no other place for it to go.

Pg 195 CLR

8-20-02

(9.1-1) From an list of n elts use $n-1$ comparisons to find the smallest elt.



I have another $\lceil \lg n \rceil - 1$ comparisons to do. ...

?

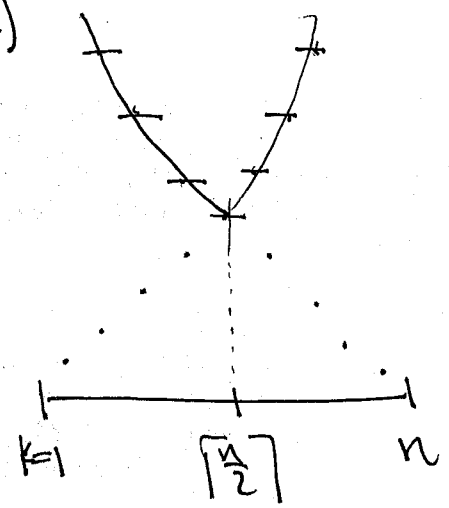
(9.1-2) ?

$$T(n) \le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))$$

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

$$E[T(n)] \le \sum_{k=1}^{n} \frac{1}{n} E[T(\max(k-1, n-k))] + O(n)$$

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k \le \lceil n/2 \rceil \end{cases}$$



Pg 188 CLR

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k \le \lceil n/2 \rceil \end{cases}$$

$$E[T(n)] \le \frac{1}{n}\sum_{k=1}^{n} E[T(\max(k-1, n-k))] + O(n)$$

$$\le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n)$$

Assum $E[T(k)] \le ck$ $\quad T(n) = O(1)$ $\quad n \le N$

$$E[T(n)] \le \frac{2}{n} c \sum$$

pg 189 CLR

9.2-1

A zero length call to Randomized-Select would be one where $q = p$ & Randomized-Select$(A, p, q-1, i)$ is called or $q = r$ & Randomized-Select$(A, q+1, r, i-k)$ is called.

From line (3) $q \in [p, r]$ & thus cannot be smaller or greater than

If $q = p$

$k = p - p + 1 = 1$

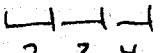
& $i < k = 1$ cannot be true since $i_{min} \equiv 1$

$\therefore$ Randomized-Select$(A, q+1, r, i-k)$ is called

If $q = r$

$k = r - p + 1$

$i < k = r - p + 1$ must be true $\forall$



$4 - 2 + 1 = 3$

i except $i = r - p + 1$ if it is true then

Randomized-Select$(A, p, q-1, i)$ is not a zero length call.

if $i = r - p + 1$ all is true then line # 5 would have

~~Randomized-Select$(A, q+1, r$~~ Stopped the execution.

(9.2-2)

one would not expect an indicator of true/false to make much difference (i.e. how the partition is split)

on the running time of the algorithm

(9.2-3) Randomized-Select(A, p, r, i)

~~while (p < r) do~~

```
lftpt = p
rgtpt = r
while (rgtpt - lftpt + 1 > 1) do
    q ← Randomized-Partition(A, lftpt, rgtpt)
    k ← q - lftpt + 1
    if (k = i)
        return A[k]
    else
        ~~lftpt = p~~
        ~~rgtpt = q-1~~
        if (i < k)
        /* lftpt = ~~p~~ lftpt */
            rgtpt = q-1
        else
            lftpt = q+1
        /* rgtpt = rgtpt */
        endif
endif.
```

Pg 189 CLR

9.2-4

$A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$

1) Call Randomize-Select w/ $i = 1$. to find the minimum.

q ← Randomized-Partition ← 10 or 9 it returns 10 see below

$A = \langle 3, 2, 1, 0, 7, 5, 4, 8, 6, 9 \rangle$ produces

10
9

$k = 9 - 1 + 1 = 10$

partition returns the index into Array A called q → with indices all elts $\leq$ q are in the left partition & all elts $\geq$ q in the right partition

9

Call Rand-Sel(A, 1, 9, 1)

q ← 8 in worst case

$A = \langle 3, 2, 1, 0, 7, 5, 4, 6, 8 \rangle$

$k = 9 - 1 + 1 = 9$

Call Rand-sel(A, 1, 8, 1)

⋮

Pg 191 CLR

8-21-02 /

$$3\left(\left\lfloor \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rfloor - 2\right) = 3\left\lfloor \frac{n}{10} \right\rfloor - 6 > \frac{3n}{10} - 6$$

$$T(n) \leq \frac{cn}{5} + c + \frac{7cn}{10} + 6c + an$$

$$= \frac{2cn}{10} + \frac{7cn}{10} + 7c + an$$

$$= \frac{9cn}{10} + 7c + an$$

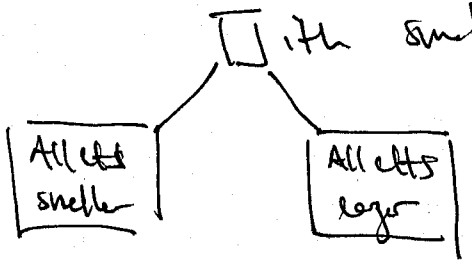
$$= cn + \left(-\frac{cn}{10} + 7c + an\right)$$

9.3-1 ?

9.3-2 $n \geq 140$?

9.3-3 One would use this notion to find the median of each input set & partition around that elt.
Since we can find the median in $O(n)$ amount of work its use in the quick sort algo does not change its running time.

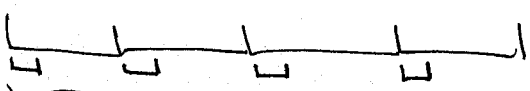
9.3-4 Consider a ~~binary tree~~ ~~OR~~ ~~OS~~ ~~heap~~ ~~stored~~
Then given the ith smallest do the comparisons that were done to determine that this elt was the ith smallest & construct a binary tree as follows



We know that we have to ~~have~~ be able to ÷ this tell every up like this for it not ~~then~~ ~~for~~ ~~the~~ (i.e. there is some elt whose relationship we can't determine from the past results) this unrelatin elt ~~must~~ could be slightly less

then the ith smallest, bumping it out of its spot. Thus we can partition the set into 2 sets of size $i-1$ + $n-i$

9.3-5 Use this black-box to do the partitioning in the routine Randomized-Select rather than Randomized-Partition

9.3-6  ?

m

$= \lfloor \frac{n}{k} \rfloor$

9.3-7 Find the median in $\Theta(n)$ work.
Subtract the median from each elt in $\Theta(n)$
take the absolute value of each elt. in $\Theta(n)$
return the kth smallest #'s.

9.3-8 ?

9.3-9 The pipeline should be located at the median of the distances from each oil well. Since we can determine the median in $\Theta(n)$ on that axis

(9-1) Sorting requires $\Theta(n \lg n)$ in best case

(a) Selecting the ith largest req $\Theta(i)$

$= \Theta(n \lg n) + \Theta(i)$

(b) Building a max-priority que requires $O(n)$ time

heap extract max requires $O(\lg n)$ time for a max-heap of size n.

Thus total time to return top i sorted #'s would be

$$O(n) + \underbrace{\sum_{j=0}^{i-1}} O(\lg(n-j))$$

↑ to build the max-heap

↖ to [illegible] cell extract max i ## times.

$$= O(n) + O\left(\sum_{j=0}^{i-1} \lg(n-j)\right) = [illegible] + O(n) + o(i \lg n)$$

little o.

= [illegible] use fact lg is a monotonically increasing function of its input to bound this summation by an integral.

(c) Order statistics can be found in $\Theta(n)$ time

Partitioning requires $\Theta(n)$ time pg 148 CLR

+ sorting the top partition of ~~the~~ ; elts requires $\Theta(i)$

Thus the total time ~~to sort~~ would be $\Theta(n) + \Theta(i)$.

9-2

(a) If x_k is the mean then assuming n is odd

$$\sum_{x_i < x_k} w_i = \sum_{x_i < x_k} \frac{1}{n} = \frac{1}{n}\left(\frac{n+1}{2}\right) = \frac{1}{2}\left(1 - \frac{1}{n}\right) < \frac{1}{2} \quad \checkmark$$

$$+ \sum_{x_i > x_k} w_i = \frac{1}{n}\frac{(n-1)}{2} = \frac{1}{2}\left(1 - \frac{1}{n}\right) < \frac{1}{2} \quad \checkmark$$

(b) Sort the elts x_i in $O(n \lg n)$ time, & their corresponding w_i's using the same permutation. Then form a cumulative sum of w_i's. When this ~~sum~~ ~~for~~ sum becomes greater than $1/2$ stop. The previous elt is the weighted (lower) median.

(c)

$$\underbrace{\begin{pmatrix} x_1 \\ w_1 \end{pmatrix} \begin{pmatrix} x_2 \\ w_2 \end{pmatrix} \cdots \begin{pmatrix} x_{k-1} \\ w_{k-1} \end{pmatrix}}_{} \begin{pmatrix} x_k \\ w_k \end{pmatrix} \underbrace{\cdots \begin{pmatrix} x_n \\ w_n \end{pmatrix}}_{}$$

?

(c)

w_i	$1/6$	$1/2$	$1/3$
x_i	3	1	2

$\frac{1}{6} + \frac{1}{2} + \frac{1}{3}$

$\frac{1}{6} + \frac{3}{6} + \frac{2}{6} = 1$

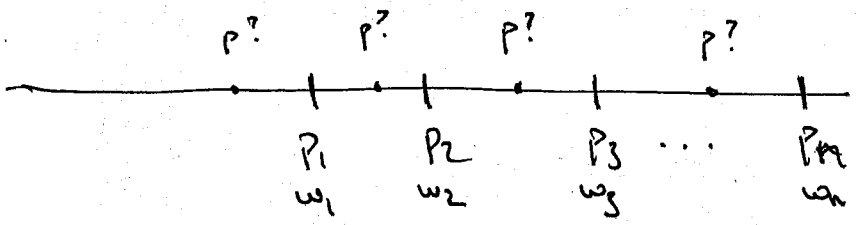
$\bar{w}_i$	$1/2$	$1/3$	$1/6$
$\bar{x}_i$	1	2	3

median of $w = 1/3$ $\quad x_k = 1$

median of $x = 2$

?

(d)



Let p^* = pt given by the weighted median of p_i w/ weights w_i.

?

(e) ?

pg 198 CLR

8-21-02 I

9-3

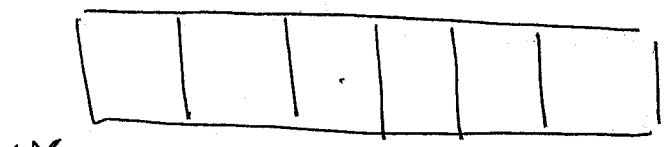
(a) ?

(b) $i < n/2 \quad T_i(n) = \lfloor n/2 \rfloor + T_i(\lceil n/2 \rceil) + T(2i)$

?

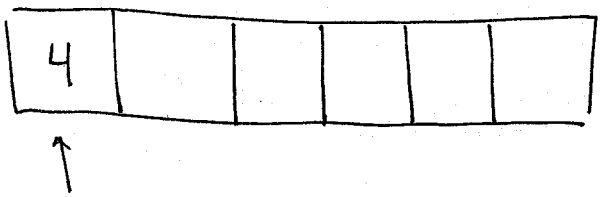
pg 203 CLR

10.1-1

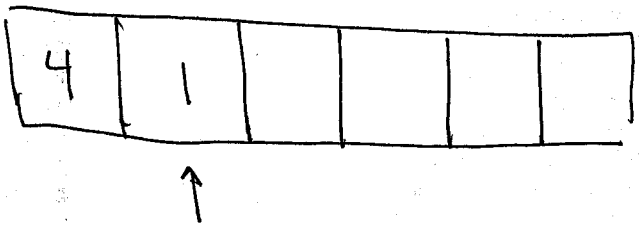


top(S) = 0

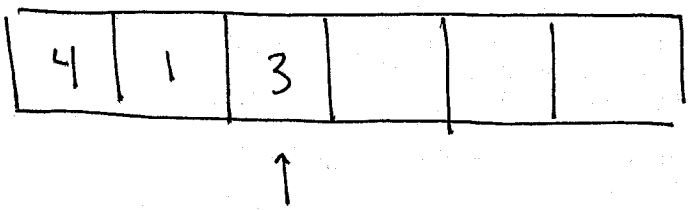
push(S,4)



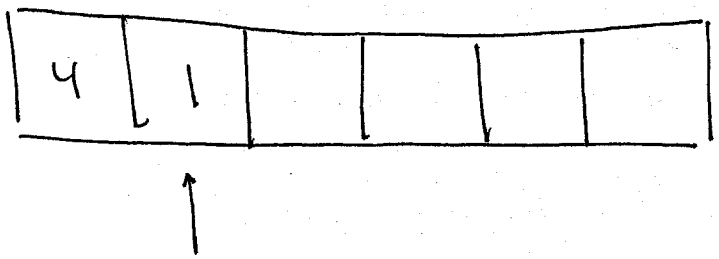
push(S,1)



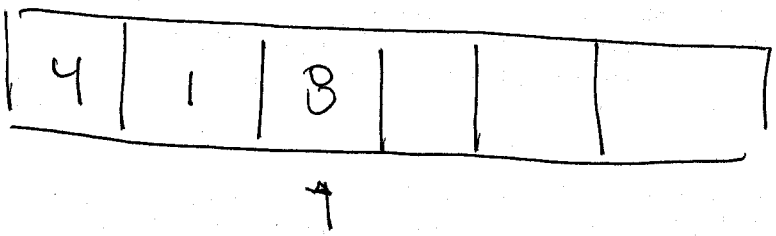
push(S,3)



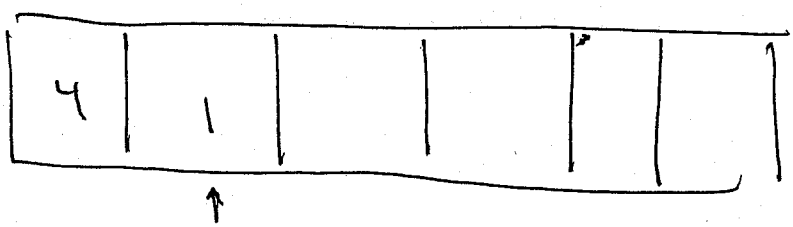
pop(S)



push(S,8)



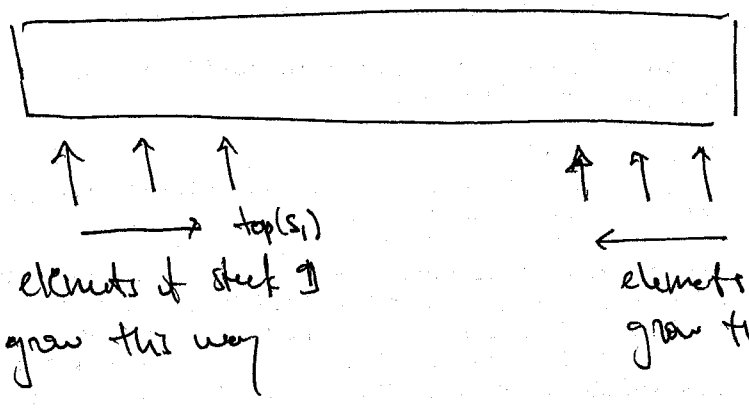
pop(S)



(10.1-2)

1

A:



Stack-Empty(S)

if top(S) = 0

~~return~~ true

~~else~~

~~return false~~

initially

$top(S_1) = 0$

$top(S_2) = n+1$ w/ ~~0~~ one based indexing.

push(S, x)

if $S == S_1$

$top[S_1] \leftarrow top[S_1] + 1$

$S_1[top[S_1]] = x$

else

$top[S_2] \leftarrow top[S_2] - 1$

$S_2[top[S_2]] = x$

end

pop(S, x)

~~if $S = S_1$~~

~~if stack-empty~~

if stack-empty(S)

error(stack underflow)

else

if $S == S_1$

$top[S_1] \leftarrow top[S_1] - 1$

return $S_1[top[S_1] + 1]$

else

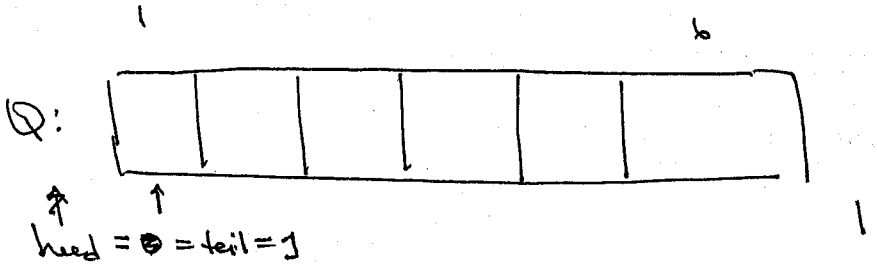
$top[S_2] \leftarrow top[S_2] + 1$

return $S_2[top[S_2] + 1]$

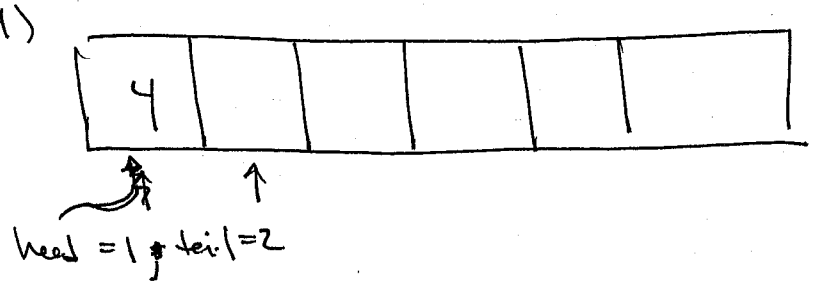
end

end

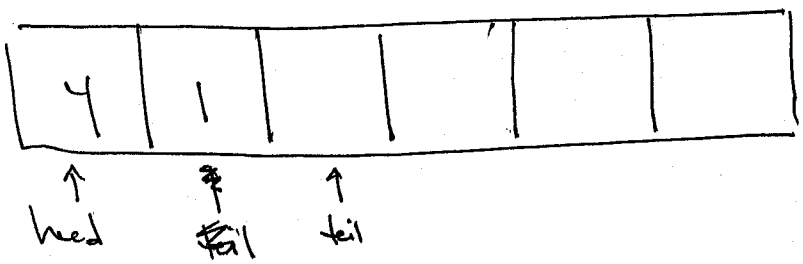
10.1-3



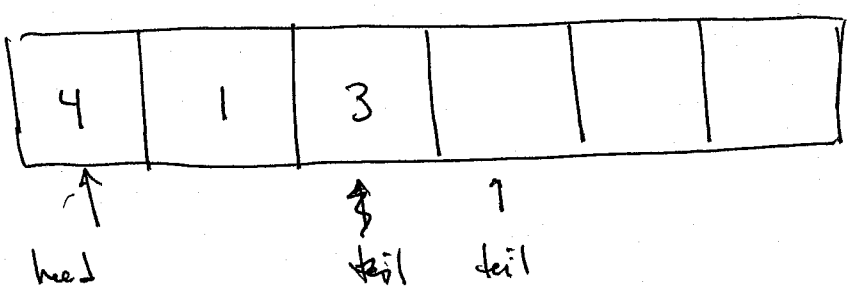
enque(Q,4)



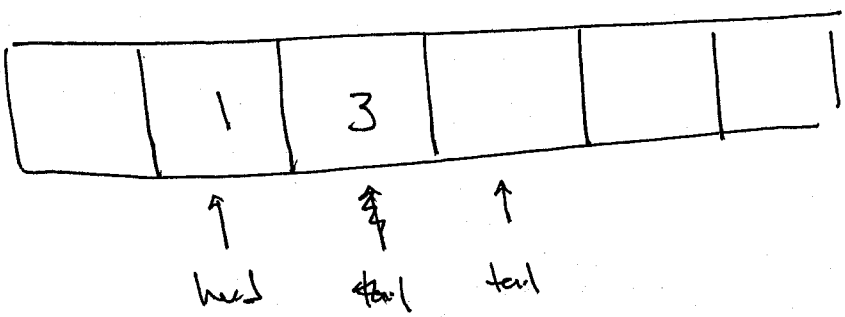
enque(Q,1)



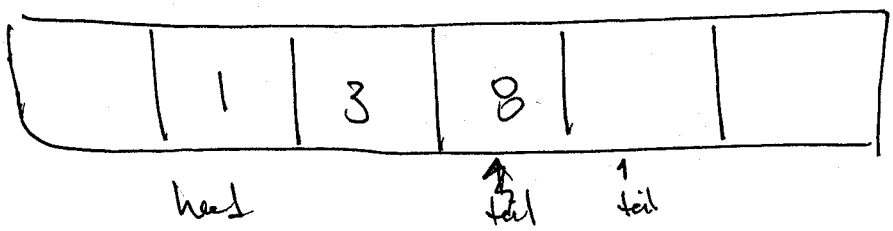
enque(Q,3)



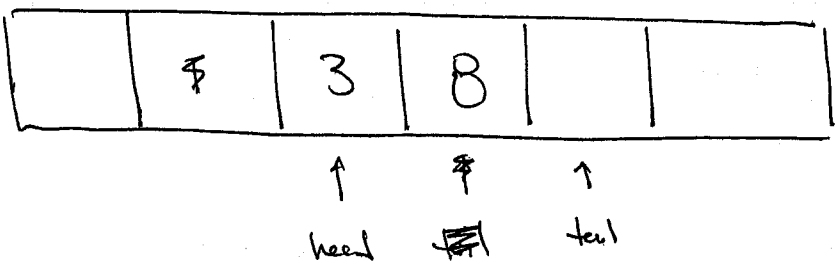
dequeue(Q)



enque(Q,8)

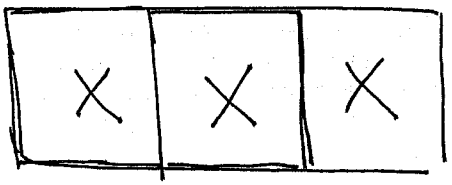


deque(Q)



(10.1-4) overflow happens when

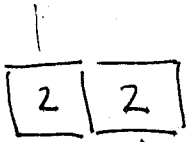
tail + 1 = head



head = tail

looks like this represents a full queues

head
↑
tail



h = t = 1 t = 2 t = 3

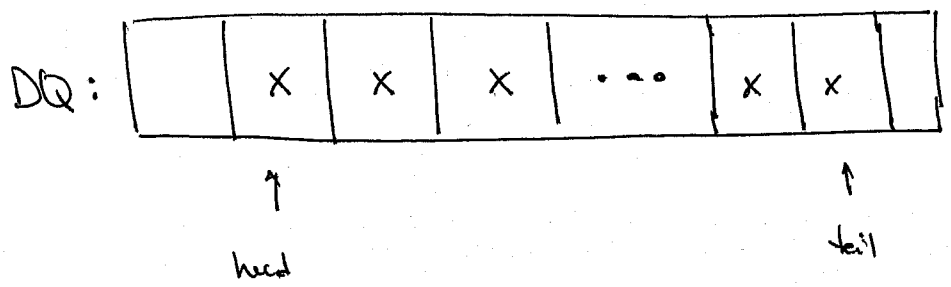
Thus Enqueue(Q, x) would give:

```
if (head(Q) == tail(Q) + 1) th ⟸ don't fully understand this statement!!
    error('queue full');
endif
Q[tail[Q]] ← x
if (tail[Q] = length[Q])
    then tail[Q] ← 1
```

$$tail[Q] = tail[Q] + 1$$

```
Dequeue(Q)
    if (head(Q) = tail(Q))
        error('queue underflow');
    exit
end.
```

10.1-5 ~~Enqueue~~ deque

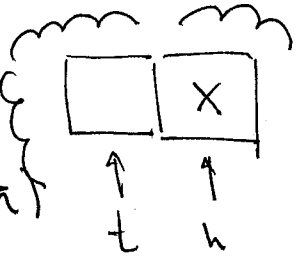


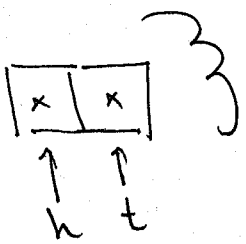
initially $head = tail = 1$. ; overflows when $tail + 1 \text{ modulo } n = head$ or $head - 1 \text{ modulo } n = tail$.

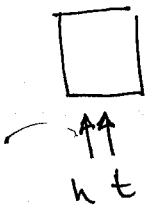
```
enqueuehead(DQ, x)
    if (head - 1 mod n == tail)
        print overflow.
    exit
    head = (head - 1) mod n
    DQ[head] ← x
end
```

```
enqueuTail(DQ, x)
    if (tail + 1 n == tail)
        print overflow
    end
    tail = (tail + 1) mod n.
    DQ[tail] = x
end
```

```
dequeHead(DQ, x)
    if ( (head-1) mod n = (tail+1) mod n )
        exact underflow )
    // Add underflow check here */
    head = (head+1) mod n
    x ← DQ[head]
    return x
```



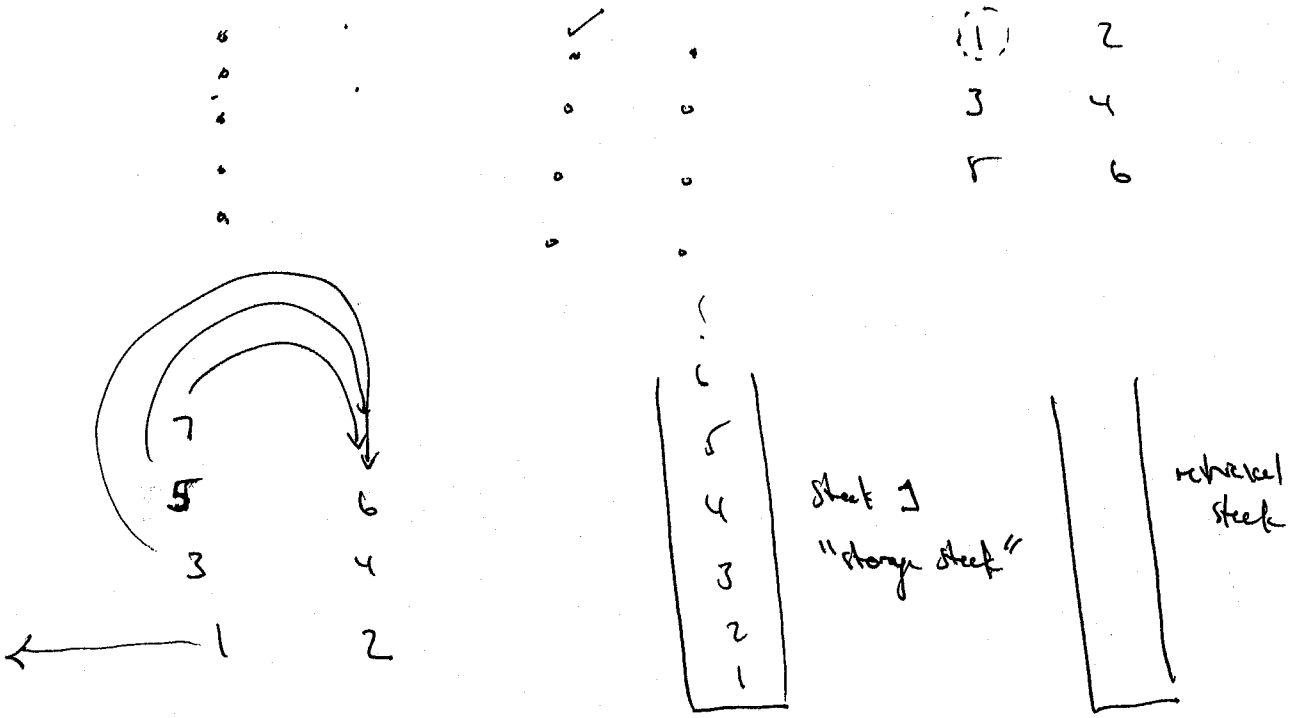





```
dequeTail(DQ)
    x ← DQ[tail]
    tail = (tail-1) mod n
    return x
```

10.1-6

queue using 2 stacks, say S_1 & S_2

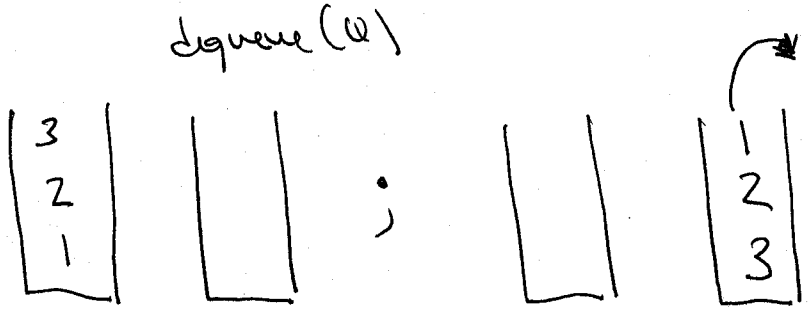


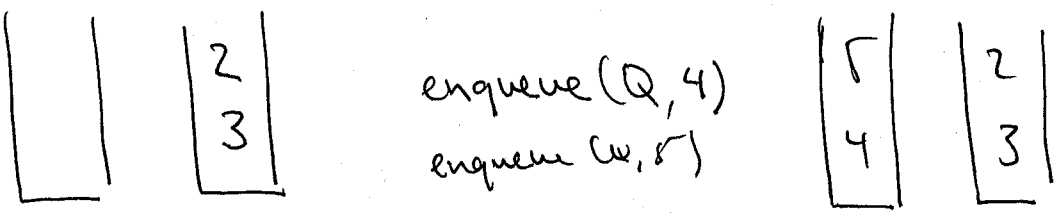
While storing you fill the storage stack. While retrieving you empty the retrieval stack & when empty you then fill it w/ all elts from the storage stack.

to ~~dequeue~~ enqueue(Q,1), ~~deq...~~ enqueue (Q,2), ~~deq...~~ enqueue (Q,3)...

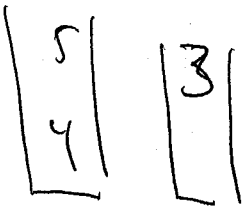
dequeue(Q)

results in





dequeue(Q)



Thus the pseudocode is the following:

```
enqueue(S1, S2, x)
    push(S1, x)
end
```

```
dequeue(S1, S2, x)
#
if (Stack-Empty(S2))
    /* copy all
    if (Stack-Empty(S1))
        error(queue empty)
    else
        /* pop all elts off S1 &
           onto S2 */
    endif
endif
return(pop(S2))
```

~~xxx~~

running tim of enqueue is $O(1)$

running time of dequeue is

$O(1)$ sometimes (if $S_2 \neq \phi$)

+ $O(|S_1|)$ (the # of elts on Stack 1)

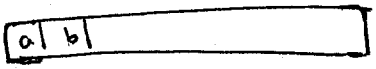
Pg 204 CLR

9-19-02

10.1-7

return c.

x, x, x,



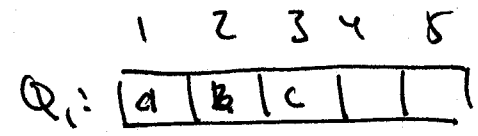
a, b, c

Idea on ~~stack~~ 2 queue is always empty & one stack is being filled up. So to push an element onto our fake stack we place it a in the last position in the non empty queue To pop the last pushed elt ~~of~~ the stack we shift all elts over to the empty queue & ~~return~~ the returning the last element.

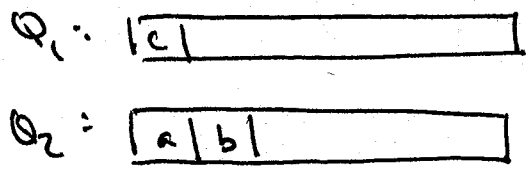
Thus the sequence push(Q_1, Q_2, a), push(:, b), push(3)

pop(), push(4) would look like

(~~After~~ 3 pushes ...)



~~After~~ to ~~pop~~ call pop once



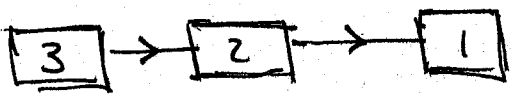
return c.

Thus as in problem 10.1-6 the pushing runs in $\Theta(1)$ time the popping runs in $\Theta(|Q_2|)$ time.

10.2-1 (1) If list-insert is simply to place element x into the list, at some known location, then Yes. The code is the same as a doubly linked list & deleting any reference to previous pointer.

(2) Delete can not be implemented in O(1) time since one needs to be able to find the elt "behind" the elt. deleted. this requires a walk of the linked list from the beginning to the elt deleted.

10.2-2



push(S,x)

~~[illegible]~~

list-insert(L,x)

pop(S)

list-delete(L,head(L))

10.2-3

enqueue(L,x)

list-insert(L,tail(L))

dequeue(L,x)

list-delete(L,head(L))

we have to be able to quickly O(1) find the tail for this method

(10.2-4) List-Search'(L, k)

x ← next(nil(L))

while x ≠ nil(L) + key(x) ≠ k

do x ← next(x)

return x

?? Don't know what asked.

(10.2-5)

Insert running in Θ(1)

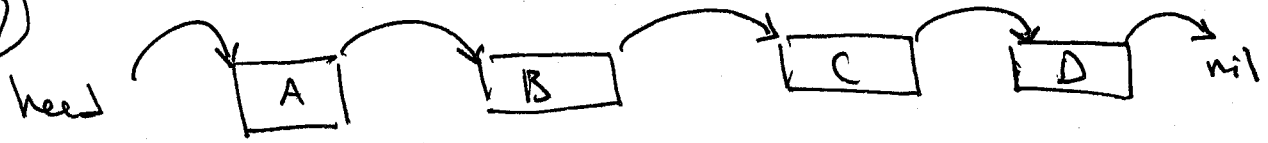
delete " " Θ(1)

Search " " Θ(n)

(10.2-6) Using a linked list one could "join" the head & the tail of one list + the tail & the other list in O(1) time

What about duplicate elements?

(10,2-7)



reverse(L)

~~// fix tail elt if O(n) sort~~

~~while (next(~~ x_previous = nil.

x = head(L)

~~while (next(x) ≠ nil)~~

~~x->~~.

next → x → next.

while (next ≠ nil)

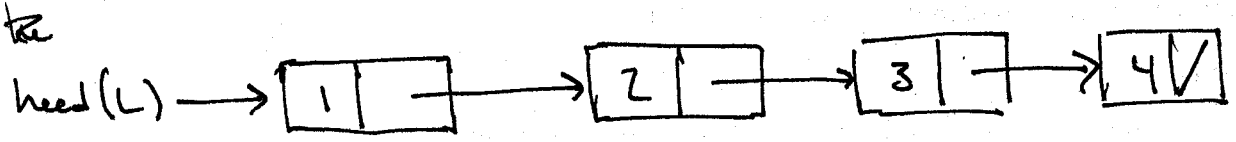
x → next = x_previous.

x_previous = x

next = next → next.

endwhile

10.2-7



head(L') = head(L)

?

10.2-8 ?

pg 212 CLR

9-22-021

10.3-1

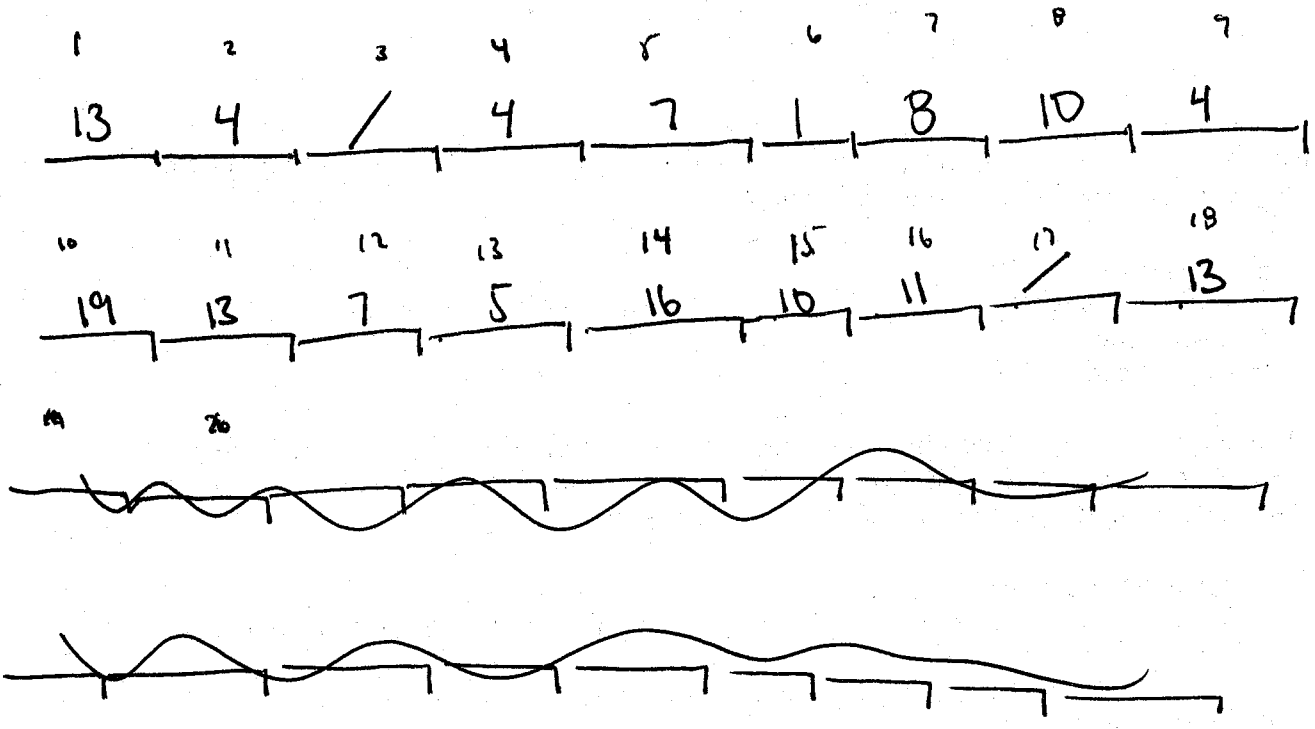
Multiple Array representation:

Array #:	1	2	3	4	5	6
next:	2	3	4	5	6	/
key:	13	4	8	19	5	11
previous	/	1	2	3	4	5

seems rather trivial ...

Single Array representation: Using convention that the keys ~~and~~ pointers are stored in the array as key, next, prev.

∴ for a 6 elt list I need an array 18 elts long.



10.3-2

Allocate-Object:

One needs to have a way of representing the available spaces. I'll assume this is done ~~the~~ in the same manner as w/ the multiple array representations with an additional linked list representing the free elts.

↑ one dimensional

Then the routines Allocate-Object + Free Object follow exactly the routines given on pg 211 + 212 of the text.

10.3-3 Because the list of freed elts is a singly linked list & there is no previous pointer. We take elts of the front of the list only.

10.3-4 I would assume that as we delete free elts we would shuffle the elts down from above into the deleted positions?

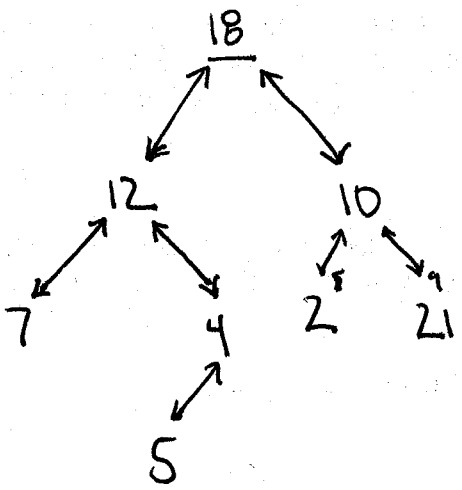
9-26-02 3

(10.3-5) Assume free list is represented by a doubly-linked list.

Don't see how to guarantee that when I put a new elt onto the free list (or allocate space for this free elt) that it will not be with in $[1, \ldots, m]$ elts.

I must be clever to loop over the free elts. ... ?

10.4-1



10.4-2 Most naive implementation is

```
Print-Node(x)                    /* x is the index
                                 if(key == nil)
                                   return.
print-key(x)
 call print-node(left(x))
 call print-node(right(x))
```

Running time is:

$T(n) = \Theta(1) + 2T(n/2)$

$T(n) = \Theta(1) + 2(\Theta(1) + T(n/4))$

$\Rightarrow T(n) = 1 + 2 + \cdots \lg(n)$

$= \sum_{k=1}^{\lg(n)} k + \Theta(1)$

By master thm:

$T(n) =$

$a = 2, b = 2$

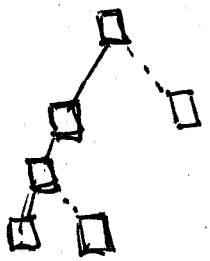
$\log_2 2 = 1$

$T(n) = O(n^{\lg \log_2 2 - \epsilon})$

$\Rightarrow T(n) = \Theta(n \lg n)$

asymptotic order.
Not the correct ~~order~~ ~~statistics~~ ...

10.4-3 ?



Not sure how to do?

print-B-Tree(H)

10.4-4 while(true)

try to walk left.
else print node
try to move pointer to 1st right sibling.
if

⋮

10.4-5 ?

10.4-6 ?

10-1

pg 217 CLR

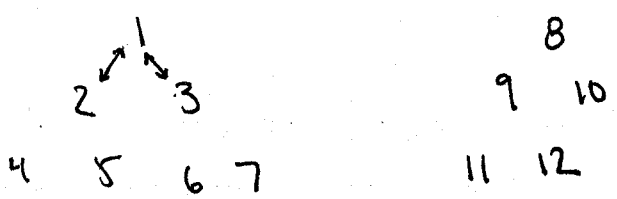
	unsorted singly linked	sorted singly linked	unsorted ~~singly~~ doubly linked	sorted doubly linked
Search(L,k)	O(n)	O(n)	O(n)	O(n)
insert(L,x)	O(n)	O(n)	O(1)	O(1)
delete(L,x)	O(n)	O(n)	O(1)	O(1)
Successor(L,x)	O(n)	O(1)	O(n)	O(1)
predecessor(L,x)	O(n)	O(n)	O(n)	O(1)
minimum(L)	O(n)	O(1)	O(n)	O(1)
maximum(L)	O(n)	O(1)	O(n)	O(1)

* ~~Note~~: ~~F~~ ~~OR~~ ~~ASSUMING~~ ~~SUCCESS~~

* Note: successor returns the next elt in the sorted array.

(10-2) implement mergeable min-heap w/ linked list.

1 2 3 4 5 6 7 8 9 10 11 12



A mergeable heap can be represented by a binary tree which are represented in section 10.4.

Note: All heap building takes place in $\Omega(n)$ time since every elt of the heap must be touched.

(a) Sorted lists: then ~~B~~ building takes place in $\Theta(n)$ since simply placing the ~~el~~ elts in a binary tree creates a min-heap.

~~B~~ insert requires $O(\lg n)$ time to bubble the elt down ~~then~~ ~~B~~ ~~to~~ since replacing elts requires in $\Theta(1)$ time.

minimum requires $\Theta(1)$ time
extract-min requires $O(\lg n)$
union requires $O(n_1+n_2)$ time to merge since we

basically pick elts off at each heap + create another heap.

(b) list or unsorted.

I would use a build-heap procedure which requires O(n lg n) time + then call the routine for sorted lists above.

(c) It does not matter that the dynamic sets are disjoint. Use the procedure above.

10-3

(a) Compact-list-search + compact-list-search' differ only in that # of the prime version has t iterations of guessing to increase the i iterator + then a # of linear searching to obtain the final location. ~~but both of which~~ If the unprimed version requires t iterations of the while loop then we require ~~both~~ the total # of steps in both loops to be at least t.

(b) do t random steps + then $E[X_t]$ more steps to get ~~the~~ to the target.

(c) $$E[X_t] = \sum_{i=0}^{\infty} Pr\{X_t \geq i\} = \sum_{i=0}^{n} Pr\{X_t \geq i\}$$

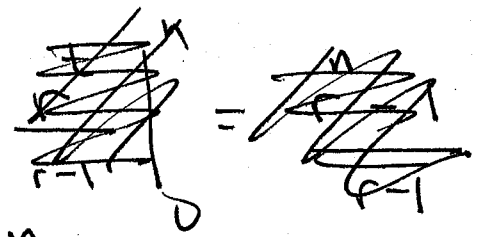
w/ $Pr\{X_t \geq i\} = \frac{n-i}{n}$

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

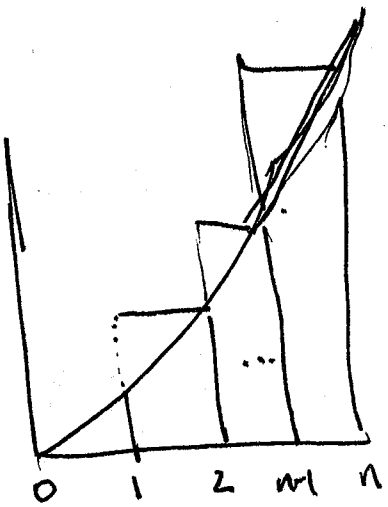
$$E[X_t] = \sum_{i=0}^{n} \left(1 - \frac{i}{n}\right)$$

? Ass

$$E[X_t] \le \sum_{r=1}^{n} \left(1 - \frac{r}{n}\right)^t$$

(d) ✓ $\sum_{r=0}^{n-1} r^t =$ 

$$\le \int_0^n r^t \, dr$$



$$\frac{r^{t+1}}{t+1}\Big|_0^n = \frac{n^{t+1}}{t+1}$$

(e) $E[X_t] \le \sum_{r=1}^{n} \left(1 - \frac{r}{n}\right)^t \le$ ✱

$$0^t + \left(\frac{1}{n}\right)^t + \left(\frac{2}{n}\right)^t + \cdots + \left(\frac{n-1}{n}\right)^t$$

$$= \frac{1}{n^t} \sum_{r=0}^{n-1} r^t \le \frac{1}{n^t} \frac{n^{t+1}}{(t+1)} = \frac{n}{t+1}$$

(f) from (d) & (b) $O(t + \frac{n}{t})$ time

(g) One would optimize on t to produce the optimal Algo

$$1 - \frac{n}{t^2} = 0 \rightarrow t = \sqrt{n}$$

$\therefore$ $O(\sqrt{n})$ for the optimal Algo.

Why is this the running time of compact list search?

(h) ?

11.1-1 Loop over each element in the table + find the maximum element. $\Theta(n)$ is worst case

11.1-2

1 $\Leftrightarrow$ element is present

0 $\Leftrightarrow$ element is not present.

11.1-3 ?

11.1-4 ?

$$X_{ij} = I\{h(t_i) = h(t_j)\}$$

$$E[X_{ij}] = \frac{1}{m}$$

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n} 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{ij}]$$

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

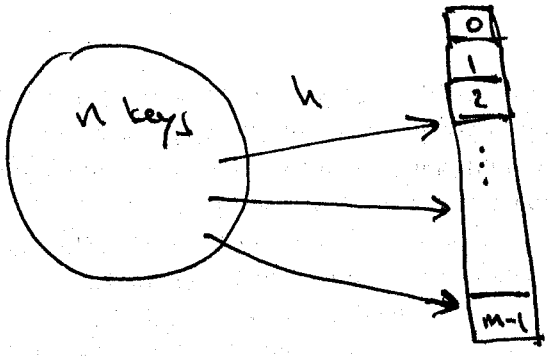
$$= 1 + \frac{1}{nm}\left[n^2 - \frac{1}{2}(n)(n+1)\right]$$

$$= 1 + \frac{1}{nm}\left[n^2 - \frac{1}{2}n^2 - \frac{n}{2}\right] = 1 + \frac{1}{nm}\left[\frac{n^2}{2} - \frac{n}{2}\right]$$

$$= 1 + \frac{1}{m}\left[\frac{n}{2} - \frac{1}{2}\right] = 1 + \frac{1}{m}\left(\frac{1}{2}\right)(n-1)$$

$$= 1 + \frac{n-1}{2m} = 1 + \frac{n}{2m} - \frac{1}{2m}$$

11.2-1



Expected # of collisions = ?

$$S = \{\{k,l\} : k \neq l \text{ and } h(k) = h(l)\}$$

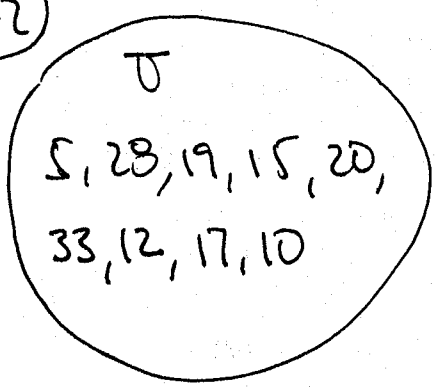
$\forall k \; \exists \; n-1$ choices for l. of these each has a prob. of hashing to spot $h(k)$ of $\frac{1}{m}$.

$\therefore$ The # that has to spot $h(k)$ is $\frac{(n-1)}{m}$

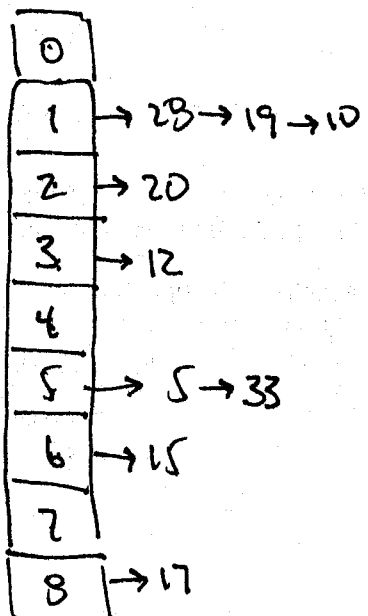
$\therefore$ The expected # of elements to collide is this # averaged over every element in U.

$$E[\#\text{ collisions}] = \frac{1}{n}\sum_{n}\left(\frac{n-1}{m}\right) = \frac{n-1}{m}$$

11.2-2



$\therefore$ $h \equiv k \bmod 9$



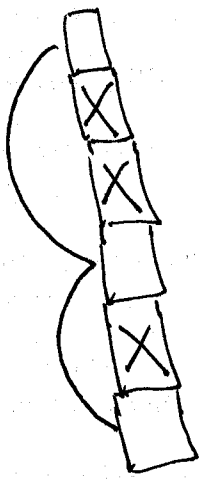
11.2-3 insertions thru ~~reqr~~ require us to find the location in the list where the key should be inserted. This involves on average of a binary search $\lg(\alpha) = \lg(n/m)$ time.

also

Successful searches + unsuccessful searches require $\Theta(\lg(\alpha))$ time

deletions require the time to find the element + then a constant time to delete it.

11.2-4 ?

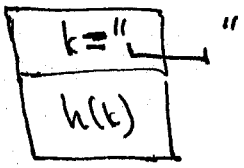


11.2-5 $|U| > n \cdot m$ Then one slot must have n keys for if every slot had $\leq n$ elements the total # of elements stored would be $\leq n \cdot m < |U|$ Thus pick the slot that ~~# with #~~ contains at least ~~#~~ n elements. The elements from the

universe that map to that spot form our subset that we are looking for.

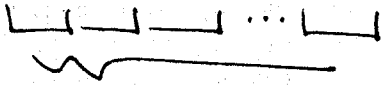
pg 236 CLR 02-14-03

11.3-1



we could sort the linked list by hash values h(ki) & when searching for a given key k* use the binarian algorithm to search for h(k*)

11.3-2 ÷ method $h(k) = k \bmod m$.



[m] = 32 bit computer word.

[k] >> 32 bit computer word

I believe k mod m is the ~~test~~ best sorter ~~th~~ of bits/ words in the number k Just computing those rather than the entire integer k will produce a quicker & ~~fast~~ less space requiring algorithm.

11.3-3 $[x] = \_\,2^p + \_\cdot 2^{p-1} + \_\,2^{p-2} + \cdots + \_\,2 + \_$

? Don't follow

11.3-4 $m = 1000$ $h(k) = \lfloor m(kA \bmod 1) \rfloor$

$A = \frac{\sqrt{5}-1}{2} =$

$61, 62, 63, 64, 65 \longrightarrow ?$

$kA \bmod 1$ = take fractional part of kA

$h(61) = 700$

$h(62) = 318$

$h(63) = 936$

$h(64) = 554$

$h(65) = 172$

(11.3-5) $\Pr\{h(k) = h(\ell)\} \leq \epsilon$

ϵ-universal $\mathcal{H}$

?

(11.3-6) $Z_p = \{0, 1, 2, 3, \cdots, p-1\}$

$h_b : D \to B \qquad b \in Z_p$ on input n-tuple

$$h_b(\langle a_0, a_1, \cdots, a_{n-1}\rangle) = \sum_{j=0}^{n-1} a_j b^j$$

$$\mathcal{H} = \{h_b : b \in Z_p\}$$

$$\Pr\{h(k) = h(\ell)\} = \Pr\{h_b(k) = h_b(\ell)\}$$

$$= \Pr\left\{\sum_{j=0}^{n-1} a_j b^j = \sum_{j=0}^{n-1} \tilde{a}_j b^j\right\} \qquad ? \quad \text{How do?}$$

X = # of probes until an unsuccessful search

A_i = there is an i^{th} probe to an occupied spot.

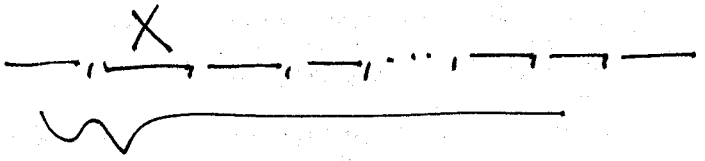
$\Pr\{X \geq i\} \leq \Pr\{A_1 \cap A_2 \cdots A_{i-1}\}$

$\Pr\{A_1 \cap A_2 \cdots \cap A_{i-1}\} = \Pr\{A_1\} \Pr\{A_2 | A_1\} \Pr\{A_3 | A_1 \cap A_2\} \cdots$

$\Pr\{A_{i-1} | A_1 \cap A_2 \cap \cdots A_{i-2}\}$.

$\Pr\{A_1\} = \frac{n}{m}$

$\Pr\{A_2 | A_1\} =$



n slots, 1 filled for sure
+ n total are filled

$\Pr\{A_2 | A_1\} = \frac{n-1}{m-1}$

pg 244 CLR

$$\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{dx}{x} = \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) \qquad \alpha = \frac{n}{m}$$

$$= \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

$\alpha = 1 \rightarrow$ hash table is full.

$\alpha = .9 \qquad E[X] = \frac{1}{.9}\ln\left(\frac{1}{.1}\right) = \frac{10}{9}\ln\left(\frac{1}{1/10}\right) = \frac{10}{9}\ln(10)$