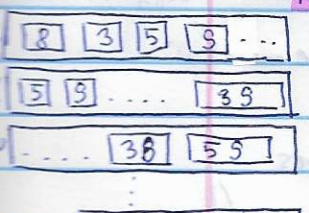
ECTURE 3
8/30/18
2.4, 2.5, 2.6

- MERGE SORT - RECURSIVE
  - split list into two halves
  - recursively sort each half
  - merge the two sorted sublists

```
mergesort(a[1,...,n])
  if n>1:
    return merge(mergesort(a[1,..., n/2]), mergesort(a[n/2+1,...n]))
  else:
    return a
```




```
merge(x[1,...k], y[1,...l])
  if k=0:
    return y[1,...l]
  if l=0:
    return x[1,...k]
  if x[1]≤y[1]:
    return x[1] ∘ merge(x[2,...k], y[1,...l])
  else:
    return y[1] ∘ merge(x[1,...k], y[2,...l])
```

→ constant amount of work per recursive call
↳ runtime $O(k+l) = O(n)$ linear

concatenation

→ runtime of mergesort

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

→ everything else but recursion
→ in this case, cost of merge()

$\frac{a}{b^d} = \frac{2}{2^1} = 1$

$O(n^1 \log_2 n)$ ↳ balanced tree

$= O(n \log n)$

- MERGE SORT - ITERATIVE

```
mergesort(a[1,...n])
  Q = [ ]   → empty queue
  for i=1 to n:
    inject(Q, [a_i])
  while |Q|>1:
    inject(Q, merge(eject(Q), eject(Q)))
  return eject(Q)
```

inject - adds element to end of queue
eject - removes and returns element at the front of queue

$n!$ —1 comp→ $\geq \frac{n!}{2}$ —1 comp→ $\frac{n!}{4}$ … $\geq \frac{n!}{2^i} = 1$

size of set

happens when $i \geq \log_2(n!) \geq \log_2\left(\left(\frac{n}{e}\right)^n\right)$

$= n\log n - n\log e$

$= \Omega(n \log n)$

$S_1 \cup S_2 = S_0$

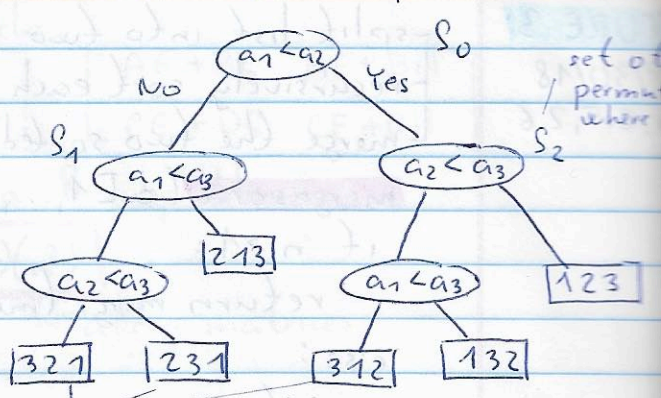
$|S_1| + |S_2| = |S_0|$

→ either $|S_1|$ or $|S_2|$ is at least $\frac{|S|}{2}$

set of

S_0 … all permutations

- n log n LOWER BOUND FOR SORTING
  - depict as tree
  - depth
    - number of comparisons on the longest path from root to leaf
    - → mergesort is optimal
      - $\Omega(n \log n)$ comparisons necessary for sorting n elements



set of permu… where

→ sorting algorithm must find … unique permutation π $a_{\pi 1} < \ldots <$

- merge sort is optimal among comparison sorts

- LEMMA
  - any comparison sort must use $\Omega(n \log n)$ comparisons
  - tree has (at least) $n!$ leaves because there are $n!$ elements = $n!$ permutations
  - binary tree of depth d has at most 2^d leaves
    - → depth of tree and time complexity of algorithm must be at least $\log(n!)$
  - note that $\log(n!) \geq cn\log(n)$
    - because $n! \geq \left(\frac{n}{2}\right)^{n/2}$ because $n! = 1 \cdot 2 \cdots n$ contains at least $\frac{n}{2}$ factors larger than $\frac{n}{2}$

    $\log(n!) \geq \frac{n}{2}\log\left(\frac{n}{2}\right)$
  - → we have established that comparison tree that sorts elements must make at worst case $\Omega(n \log n)$ comparisons
    - → merge sort is optimal
  - note
    - this argument applies only to algorithms that use comparisons
    - some sorting strategies can under certain conditions sort in linear time
      - when elements are integers that lie in a small range
      - ex. radix sort
        - runs in time $O(w \cdot n)$ where w is the number of bits to represent a_i's

- if you have a lot of items, it would cost you quite a bit to compare them
  - → better radix sort
- when item comparisons are very cheap
  - for example when comparing only first 2 bits of numbers
  - → better mergesort

- **MEDIAN**

- single typical value
- → less sensitive to outliers

- median - the element that is bigger than half and smaller than the other half
  - ↳ if the list has even length, pick the smaller of 2 middle elements

⇒ How to find median?

→ straight forward way $O(n \log n)$

1. sort
2. pick the middle

→ we will solve more general problem

- **FIND K^{TH} SMALLEST ELEMENT**

selection:

* input: list S, index K
* output: K-th smallest element in S

⇒ **RECURSIVE APPROACH**

- pick (somehow) $a \in S$ and split S into these lists

$S_L = \{$elements in S smaller than $a\}$
$S_a = \{$elements in S equal to $a\}$
$S_R = \{$elements in S bigger than $a\}$

} we can find it in $O(n)$

select (S, K): → K^{TH} element

- pick $a \in S$
- construct S_L, S_a, S_R
- if $K \leq |S_L|$: select (S_L, K)
- if $|S_L| < K \leq |S_L| + |S_a|$: return a
- if $|S_L| + |S_a| < K$: select $(S_R, K - |S_L| - |S_a|)$

- S_L, S_a, S_R can be computed in linear time

- example

$S = 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1$

$S_L = 2, 4, 1$
$S_a = 5, 5$
$S_R = 36, 21, 8, 13, 11, 20$

} when $a = 5$

⇒ What if we are unlucky and pick a as the largest element?
→ then recursion will make us pay
$n+(n-1)+(n-2)+\ldots = \Theta(n^2)$
↳ worst case scenario

⇒ Why is random $a \in S$ good
- with high probability (50%) a will be between 25% and 75% of values
- if a is 25% and 75% of S, then $\max\{|S_e|, |S_i|\} \leq \frac{3}{4}|S|$
→ discarding 25% of elements

expected time: $ET(n) = ET\left(\frac{3}{4}n\right) + O(n) = O(n)$
↳ not worst-time
- because there is randomness
- lies between $O(n)$ and $\Theta(n^2)$

• LEMMA
- on average a fair coin needs to be tossed two times before heads is seen
⇒ proof
E - expected number of tosses before head is seen
$E = 1 + \frac{1}{2}E$ which works out to $E = 2$

• QUICKSORT
- splits array exactly like median algorithm
- subarrays are sorted by two recursive calls
- worst performance: $\Theta(n^2)$
- average case: $O(n \log n)$
- empirically outperforms other sorting algorithms