Sorting Lower Bound:

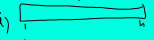
1. **Theorem**: Any comparison-based sorting algorithm takes time $\Omega(n \lg n)$ to sort n distinct items, in the worst case.
2. Hashing & hashtables

Proof of the theorem [information-based lower bound].

A **comparison-based** sorting algo is one that can only ask questions of the form, "is $A[k] < A[l]$?" for some indices k, l.

Assume items are stored in $A[1 \ldots n]$ and are comparable.

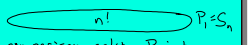
Input: A: 

Output: A: (sorted) 

To sort correctly the algo M must behave differently for different permutations (orderings).

There are $n!$ many permutations of n distinct items.

Initially, all $n!$ permutations are consistent with the information the algo has gathered so far (i.e., none).



A comparison splits P_1 into two sets: $P_{1,yes} = \{$permutations consistent with a "yes" answer$\}$

$P_{1,no} = \{ \ldots \}$

Adversary: choose answer a such that $|P_{1,a}| \geq |P_{1,\text{not } a}|$

set $P_2 := P_{1,a}$

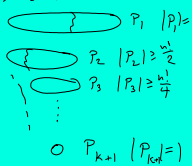
$|P_2| \geq \frac{1}{2}|P_1|$.

On 2nd comparison, adversary chooses answer a such that $|P_{2,a}| \geq |P_{2,\text{not } a}|$

$P_3 := P_{2,a}$

$|P_3| \geq \frac{1}{2}|P_2| \geq \frac{1}{4}|P_1| = \frac{n!}{4}$

and so on:



Algo asked k comparisons

By adversary's answers,

$|P_j| \geq \frac{n!}{2^j}$

$1 = |P_k| \geq \frac{n!}{2^k}$ so

$2^k \geq n!$

$k = \lg(2^k) \geq \lg(n!)$

(k = # comparisons for a worst-case input)

$$\lg(n!) = \lg\left(\prod_{j=1}^{n} j\right)$$

$$= \sum_{j=1}^{n} \lg j$$

$$\geq \sum_{j=\lceil \frac{n}{2} \rceil}^{n} \lg j \geq \sum_{j=\lceil \frac{n}{2} \rceil}^{n} \lg\left(\frac{n}{2}\right)$$

$$\geq \frac{n}{2} \lg\left(\frac{n}{2}\right) = \frac{n}{2}(\lg n - 1)$$

$$= \Omega(n \lg n) \quad //$$

Average case — assume uniformly randomly picked permutation π.

$\mathop{E}_{\text{random perm } \pi}(\text{time}) = \Omega(n \lg n)$.

Best case: $n-1$ comparisons is the fewest possible for a correct sort on any input.

Hashing & hash tables

Set U of n keys (unsigned integers)

n is large, and key values are arbitrary.

A hash table is a data structure that supports 3 basic ops:

Insert(x) — insert key x

Search(x) — search & return item with key x

Delete(x) — delete key x.

One approach: Store items in a array A indexed by the keys;

— store item with key k in $A[k]$.

Unworkable if range of keys is large. Waste of space.

Solution: Store n items in an array $A[1 \ldots m]$ for some m.

$m \ll$ range of possible keys, so store item k is at $A[h(k)]$ where

h is a function mapping keys to indices $(1 \ldots m)$ called a hash function.

h must

— be computable quickly

— be "random looking" uniform distribution of index values for the keys.

No theory of what makes a good hash function.

More heuristic — techniques hard to analyze rigorously but seem to work well in practice.

Assume the uniform simple hashing assumption: (For any key k) and index i $(1 \le i \le m)$:

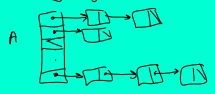
$$\Pr_k[h(k) = i] = \frac{1}{m}$$

If $m <$ # keys to insert, then by the Pigeonhole principle, $h(k_1) = h(k_2)$ for some keys $k_1 \neq k_2$. This is called a collision.

Standard ways of resolving collisions:

1. chaining
2. probing

Chaining: Each entry $A[j]$ in the hash table is a linked list of the items that hash to j $(h(k) = j)$. Collisions resolved by allowing linked lists of arbitrary length.



Expected time for Insert (= expected time for an unsuccessful search) assuming uniform hashing assumption:

$E(\text{insert}) = (\text{time to compute } h(k)) + (\text{time to traverse the list at } A[h(k)])$

$= O(1) + \Theta(\text{avg length of a list})$

$= O(1) + \Theta\left(\frac{n}{m}\right)$ — # of items / number of lists

$= \Theta\left(1 + \frac{n}{m}\right)$

under uniform simple hashing assumption, most cases don't deviate significantly from this.