CHAPTER 8 NOTES (w4-5)

└ Sorting in linear time, lower bounds for sorting

Radix sort, counting

[redacted]

Lower bounds for sorting

- ~~Q1~~: how fast can we sort?
- Q2: how fast can comparison sort be?
  - ~~Lower bounds~~
  - Only based on comparing elements. (All we've seen so far.)

- $\Omega(n)$: to examine all inputs
- All sorts so far are $\Omega(n \lg n)$
  - ↳ We'll show: this is lower bound for COMPARISON SORTING.

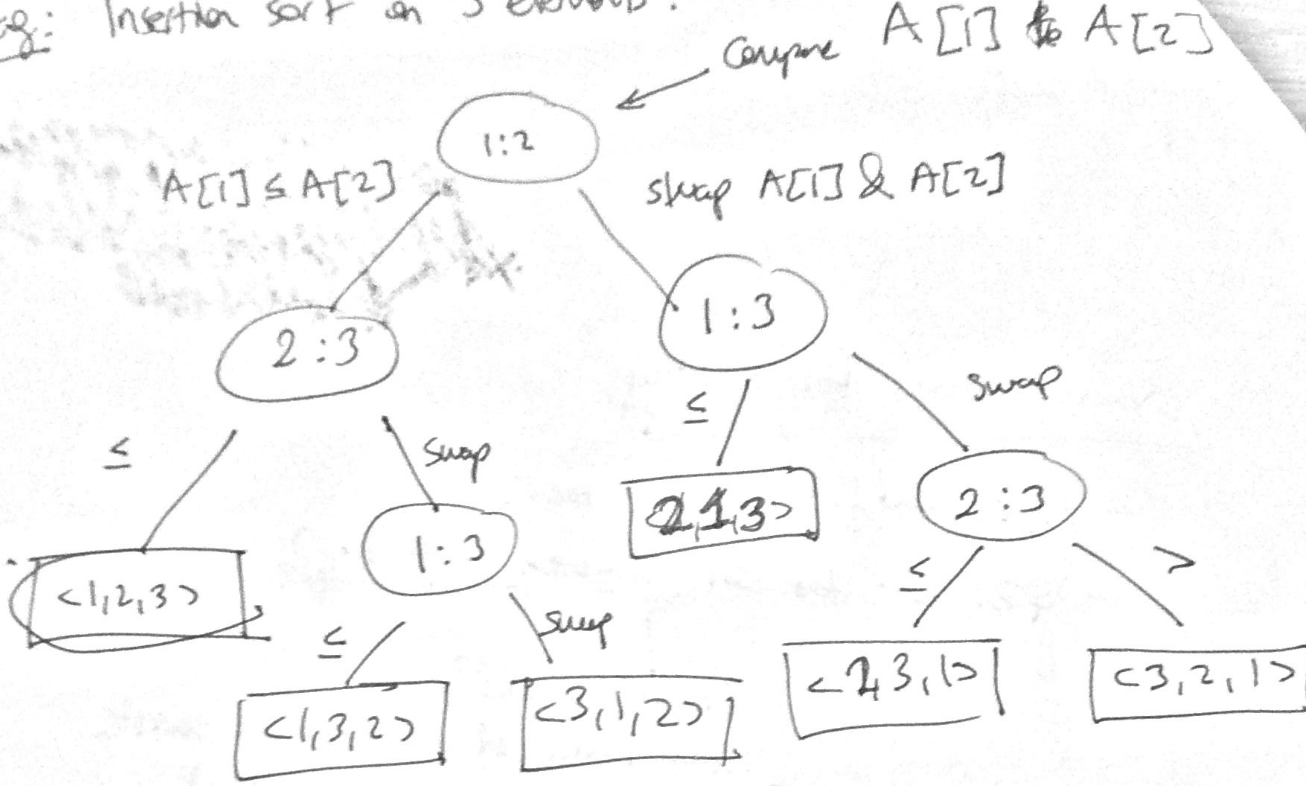
Count sort on non-integers?
- Depends on precision?
- For low-prec, that's fine.

Stirling's Form. (Approx)

$\ln(n!) \approx n \lg n - n$.

Decision tree:
- Abstraction of comparison sort.
- Abstracts away control & data movement, shows comparisons made by a particular algorithm.
- Count only comparisons

Eg: Insertion sort on 3 elements:



$\#$leaves $\geq n!$, because every permutation appears at least once.

(*) Tree models all possible execution traces.

Length of longest path from root to leaf:

- Depends on algorithm
- Insert. sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \lg n)$

Lemma: Any binary tree of height h has $\leq 2^h$ leaves.

(we all know this)

(Proof by induction.)

<u>Theorem</u>: Any dec. tree that can sort n elements must have height $\Omega(n\lg n)$

<u>Proof</u>: Tree must contain $\geq n!$ leaves, as there are $n!$ possible permutations.

* Let l: # leaves
* $l \geq n!$
* $n! \leq l \leq 2^h \Rightarrow 2^h \geq n!$

$\Rightarrow h \geq \lg(n!)$ ← Stirling approx: $n! > (\frac{n}{e})^n$

$\Rightarrow h \geq \lg(n/e)^n$

$= n\lg(n/e)$

$= n\lg n - n\lg e$

$= \Omega(n\lg n)$ □

~~Proof~~:

Corollary: Heapsort & merge sort are asymptotically optimal comp. sorts.

= SORTING IN LINEAR TIME =

* Non-comparison sorts.

Counting sort

- Assumption: numbers to be sorted are integers in {...k}

Input: A[1...n] where A[j] ∈ {1,...k}
for j = 1, 2, ..., n.

Array A and values n and k are given as parameters.

Output: B[1...n], sorted. B is assumed to be already ~~sorted~~ allocated.

Aux storage: C[...k]

Right-to-left to make it STABLE SORT.

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

- Set C[...k] to 0.
- Store number of times each integer j appears in A in C.
- C ← cumsum(C) → Now C gives the rightmost index for each integer in 1...k
- for j = n ... 1
  - B[C[A[j]]] ← A[j]
  - C[A[j]] ← C[A[j]] − 1

▷ Also show the exact algorithm on slides. AND Example

* Stable sort.

Analysis: $\Theta(n+k)$, which is $\Theta(n)$ if ~~$k = \Theta(n)$~~. $k = O(n)$ → n grows much faster than k.

→ This would imply increasing k by n which is nonsense!

⇒ SLIDES

How big a k is practical?

- 32 bit? No.
- 16 bit? Probably no.
- 8-bit? Maybe.

Q: How come counting sort is faster than $\Omega(n \lg n)$? ..

Answer: not a comparison sort.

Q: Non-integer inputs?

Answer: depends on precision.

WBm (originally w/ c, b, but I use array)

Radix sort: – IBM's original algorithm for sorting cards.

Key idea: sort by least significant digit first.

Algorithm:

```
RADIX-SORT (A, d)
for i ← 1 to d
    use a stable sort to sort array A on
    digit i.
```

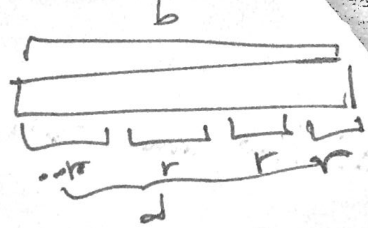
▷ Show example on slides.

Correctness: Show on slides (by induction from least to most sign. bit.)

Analysis: Assume we're using counting sort as intermediate sort. → Just counting sort

- Digits range in 1..k ⇒ each pass is $\Theta(n+k)$
- We make d passes (d: # of digits)
- $\Theta(d(n+k))$ total.
- If $k = O(n)$ ⇒ total = $\Theta(dn)$

Each number:



- n words (= ~~max~~ array length)
- b bits / word (determines max integer for RADIX SORT.)
- Use r-bit digits $\Rightarrow d = \lceil b/r \rceil$ (base for encoding?)
- Use counting sort, $k = 2^r - 1$. → MAX INT. FOR EACH COUNTING SORT CALL (ie. AUX STORAGE LENGTH)

eg, 32-bit integers, 8-bit digits

$\underline{b=32}$, $\underline{r=8}$, $d = \lceil 32/8 \rceil = 4$, $k = 2^8 - 1 = 255$.

└ # of digits, ie. # of COUNT SORT CALLS.

$$\boxed{\text{Time} = \Theta\left(\frac{b}{r}(n+2^r)\right)}$$

→ increasing r ⇒
- Fewer passes
- As $r \gg \lg n$, time grows exponentially!

Choose r to balance b/r and $n + 2^r$.

Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \boxed{\Theta\left(\frac{bn}{\lg n}\right)}$

<u>Why lg n? Answer:</u>

- If we were to choose $r < \lg n \Rightarrow \frac{b}{r} > \frac{b}{\lg n}$ and $n + 2^r$ terms doesn't ~~explode~~ improve. (SAME; $\frac{b}{r}$: ~~BOTTLENECK~~ worse)
- If we were to choose $r > \lg n \Rightarrow n + 2^r$ gets big! ⇒ (So, overall a worse choices)

eg. $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

$r < \lg n$ and $r > \lg n$ both are ~~seem to be~~ worse choices, therefore $r = \lg n$ is our best choice in general.

To sort 2^{20}-many 32-bit numbers, (~65k)

$r = \lg 2^{20} = 20$ bits, $\lceil b/\lg n \rceil = 2$ passes over all numbers. ~~per algorithm~~

→ Merge sort $\lg n = 20$ passes! (over all numbers) (at least)

<u>NOTE:</u> If we use $r = b$? (We make a single call to counting sort!)

$\Theta\left(\frac{b}{r}(n+2^r)\right) = \Theta(n + 2^b) \Rightarrow \underline{\Theta(n) \text{ if } 2^b < n} (\Rightarrow b < \lg n)$

⇒ If we know that we're dealing <u>with small numbers,</u> (Special case!) we can ~~directly~~ call COUNTING SORT, as size of aux. storage will NOT dominate.

To sum up, → (Compared to n)

- if dealing w/ "small numbers", use $r = b$, i.e., directly counting sort.

- in general, use $r = \lg(n)$, that is, choose increasingly larger basis to balance between
  - making too many counting sort calls

  AND

  - having too slow counting sort calls

How does Radix sort work faster than comparison sort?

A: - We use keys as array indices directly

(⇒ Integer assumption is the key point!)

> Q: What if lgn is larger than b?
>
> A: That is exactly the case where $r = b$ and $b < \lg n$.

BUCKET SORT (Briefly)

- Assume: input is generated by sampling from $[0,1)$ uniform dist.
- Idea:
  - divide $[0,1)$ into n equal-sized buckets, $n = \#$ inputs.
  - distribute n input into buckets
  - sort each bucket by insertion sort.
  - go through buckets, listing elements in each one.
- Analysis: ~~Average number of buckets~~
  - on average, each bucket has ≈ 1 element, by assumption.
  - ~~Each~~ insert. sort takes constant $\Theta(1)$ time.
  - in total, $\Theta(n) + n\,\Theta(1) = \Theta(n)$ time.

☆ SEE BOOK FOR A FULL ANALYSIS.