Lecture #2

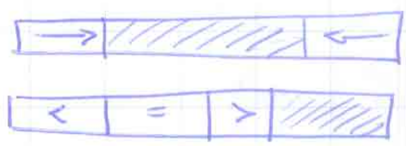
Goal: from RAM to multi-disks in sorting obj < atomic / strings

```
Quicksort (A, l, r)
   while (r-l+1 > n0)
   {
     j = pickPivotPos (A, l, r);
     swap (A[l], A[j]);
     i = Partition (A, l, r);   // pivot is in A[l] initially,
                                // A[l.. i-1] less or equal than pivot
                                // A[i, r]    more or equal than pivot

     if i < (l+r)/2 then {Quicksort (A, l, i-1); l=i; }
                    else {Quicksort (A, i, r); r=i-1; }
   }
insertionSort (A, l, r);
```

Comments:

.) Do exist various Partition procedures



.) Pivot selection makes a difference:
- random
- always first
- median out of 3
- exact median
- skewed pivot n/10

■ Last is faster because the reduced cost of "branch misprediction" (hence, less cost of flushing the pipeline) outweights the cost induced by more recursive calls: $\log_{\frac{1}{1-\alpha}} n \approx \frac{1}{\alpha} \cdot \log n$

.) Code is structured to eliminate "tail recursion", which ensures only $\log_2 n/n_0$ recursive calls at most (this was not true on classic quicksort.)

•) Use of Insertion Sort is justified by cache and small no.

<u>SELECTION</u>: The qsort-scheme can be adapted to implement a linear time (avg) procedure to select the k-ranked element.

Select (s, k)

// assert: $|s| \geq k$

pick $p \in s$ uniformly at random // First lecture

$a = \langle e \in s \mid e < p \rangle$;

if $|a| \geq k$ then return Select (a, k);

$b = \langle e \in s \mid e = p \rangle$;

if $|a| + |b| \geq k$ then return p; // this is the elem

$c = \langle e \in s \mid e > p \rangle$;

return select $(c, k - |a| - |b|)$;

Call a pivot "good" if neither $|a|$ nor $|c|$ is larger than $\frac{2}{3}n$

$\gamma = \mathbb{P}$ (pivot is "good") $\geq 1/3$ because

it must be chosen among the ones having rank $\left[\frac{n}{3}, \frac{2}{3}n\right]$.

Let $T(n)$ be the average execution time over n elements:

$$T(n) \leq cn + \gamma T\left(\frac{2}{3}n\right) + (1-\gamma)\, T(n)$$

$\rightarrow$ Solving by $T(n)$, you get $\leq 9cn$

■ Question: How do you select the smallest k-th element via one pass and $O(k)$ additional space? $\boxed{\text{Heap}}$

$\boxed{\text{Sample Sort}}$: Quicksort for Hierarchical memories and $\frac{\text{multi}}{\text{superscalar}}$ CPU

It has the same performance guarantees of multi-way mergesort but easier to code and good for multi-disks, multi-CPUs, and for strings. (MOSTLY IN-PLACE wrt mergesort)

$\boxed{\text{Key idea}}$: Instead of a single pivot, we take (K-1) splitters $s_1 \dots s_{K-1}$ which allow to break the original sequence in K pieces (buckets)

$$B_i = \langle e \in S \;/\; s_{i-1} \leq e < s_i \rangle$$

where $s_0 = -\infty$, $s_K = +\infty$

also assume that all keys are different

$\boxed{\text{Goal}}$ Select splitters so that groups are "balanced"

$\Longrightarrow$ Random Sample of $(a+1)K - 1$ elements

sort it internally (using your favorite sorter, few elems)

take $s_i = \text{Sample}[(a+1)i]$

Proof: Samples are equally spaced, and "a" is an oversampling factor which increases the probability that the splitting process was successful.

$a = 0 \rightarrow$ just the wished elems $\rightarrow$ poor partition w.h.p.

$a \simeq \frac{n}{K} \rightarrow$ exact splitters $\rightarrow$ too costly

Our following analysis show that $a = \Theta(\log K)$ is a good choice.

Taking $\simeq K \log K$ samples, buckets sizes are $\simeq \frac{n}{K}$ (avg size).

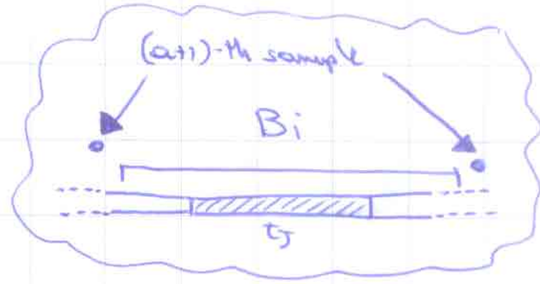
Actually $\forall i \; |B_i| < \frac{4n}{K}$ with probability $\geq 1/2$ [few executions are enough to guarantee it]

Let $s' = \langle e'_1, e'_2, \dots, e'_n \rangle$ sorted sequence

By contradiction, assume that $\exists B_i$ s.t. $|B_i| \geq \frac{4n}{K}$.

Recall that we are oversampling, so the condition above may occur even if some sampled elem lies in B_i. So we take another "street".

$$s' = \underbrace{\overset{t_J}{\boxed{2n/K \mid 2n/K \mid \;----\; \mid 2n/K}}}_{K/2 \text{ segments}}$$



If $|B_i| \geq \frac{4n}{K} \longrightarrow$ one segment is fully included in B_i
(so that $\mathbb{P}(|B_i| \geq \frac{4n}{K}) \leq \dots$)

That segment does not includes a $(a+1)$th ranked sample
[by definition of B_i]

less than $(a+1)$ samples are taken from that segment

$$\mathbb{P}(\text{sample in } t_J) = \frac{2n/K}{n} = \frac{2}{K}$$

$$\mathbb{E}[\text{samples in } t_J) = \left[(a+1)K - 1\right]\frac{2}{K} \geq \frac{3(a+1)}{2}$$

$$\mathbb{P}(\#\text{samples} < a+1) \leq \mathbb{P}\left(\#\text{samples} \leq \left(1-\frac{1}{3}\right)\mathbb{E}[\#\text{samples}]\right)$$

$$\underset{\text{chernoff bound}}{\leq} e^{-\frac{1}{9}\frac{\mathbb{E}[\cdot]}{2}} = 1/K \qquad \mathbb{P}(X \leq (1-\delta)\mu) \leq e^{-\delta^2\mu/2}$$

Applying the union bound over the $K/2$-segments we find the probability that "at least one fails, and is larger than $\frac{4n}{K}$"

$$\frac{1}{K} \cdot \frac{K}{2} = 1/2$$

SMALL, few thousands

Picking $K = \Theta\left(\min\left(\frac{n}{M}, \frac{M}{B}\right)\right)$ we obtain the optimal I/O-complexity over 1-disk. $\frac{M}{B} \approx \frac{10^9}{10^3} = 10^6$; $\frac{n}{M} \Rightarrow \frac{n}{10^9}$

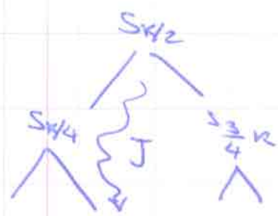
Notice that one merge-pass is enough if $n < \frac{M^2}{B} \approx \frac{10^{18}}{10^3} = 10^{15}$ → 10^3 TBs

PROBLEM : How to distribute elements in buckets using just another array, plus small additional memory.
// We do not know the bucket sizes, indeed.

$\begin{cases} 1^\wedge \text{ pass} \Rightarrow \text{compute bucket sizes} \\ 2^\wedge \text{ pass} \to \text{distribute} \end{cases}$

No in-place
but may become 'short'

$1^\wedge$ pass is interesting :



laid down as an heap

$\boxed{J = 2J \; ; \; \text{if } (\geq) \;\; J{+}{+};}$

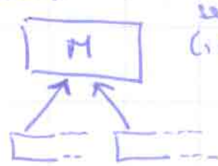
~~Predicated arithmetic on some CPUs~~ (~~Tested Itanium~~)

Predicated instruction has an additional predicate register as input and it is exec iff the boolean value in this reg is 1

} No branch misprediction, because they do not affect the instr. flow.

Multiway-mergesort The most famous (worst-case)

- Binary mergesort → 2 inefficiencies ~~problems~~ → because memory is large; After few pages Random seek ≡ Sequ. seek (if buffered)



① starts merging short runs

② When runs are very long, no help in fetching them at once in internal memory

- ① ⇒ form run of M items in one shot
  ② → multi-way merge : $k = \Theta(M/B)$

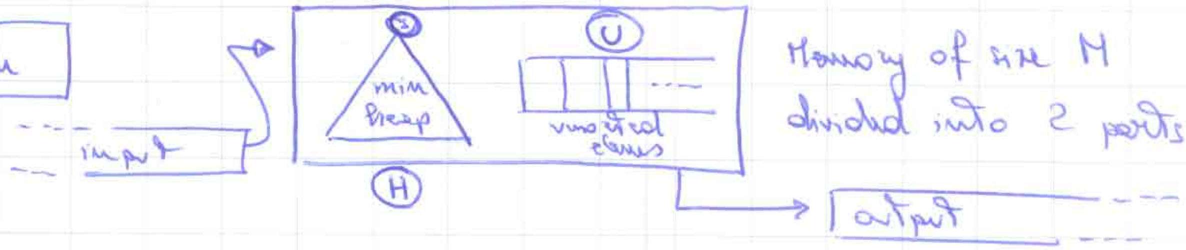
$$\#I/Os = O\left(\frac{n}{B} \cdot \log_{\frac{M}{B}} \frac{n}{M}\right)$$

•) Clearly if first-round runs are longer (say 2M) we may further reduce the tree depth.

$\boxed{\text{Snow Plow by Knuth}}$

•) We can speed up scanning via compression.

Knuth



Memory of size M divided into 2 parts

• $\forall$ step: min-heap $\Rightarrow$ output (say s)

t = next input elem

if (t > s) → add t to min-heap (run is still increasing)

else add t to unsorted elems

Prop. ① $H + U = M$ obvious

② Time $\emptyset$: $H_0 = M$, $U_0 = 0$

③ time t : $H_t = 0$, $U_t = M$ $\boxed{\text{restart}}$

↳ revolution time.

At time t we have output

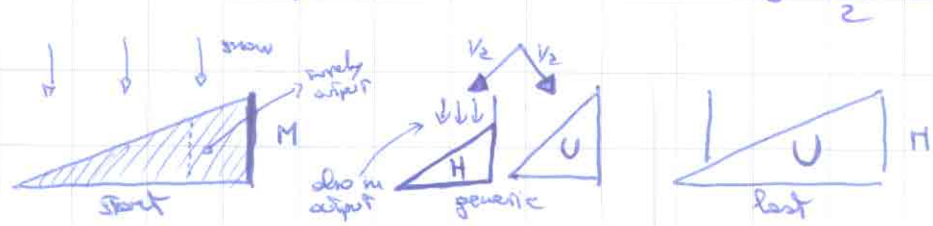
① all H_0, hence M elements

② $t/2$ elements, i.e. the ones that landed in H (assume uniform distr.)

Since $\frac{t}{2}$ landed in U too $\Rightarrow^{(\text{fill } U)}$ $\frac{t}{2} \leq M \Rightarrow t \leq 2M$

Hence all output elements are $H_0 + \frac{t}{2} \simeq M + \frac{2M}{2} = 2M$

Or



since they distribute equally and $U_t = M$ ⇓ distributed on H also = M

PERMUTING What about moving around obj?
sorting = sorted order comp + permuting

Perm < Sort in RAM $\Longleftrightarrow$ N vs $N \log N$

- This means that the larger cost here is in determining the sorted order.

On disk, we could either permute as in RAM $\equiv \Theta(N)$ I/Os or forget the given permutation and sort.
If $N > \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}$ we have that sorting from scratch is advantageous.

$\Longrightarrow$ On disk Perm $\underset{\text{may be}}{\approx}$ Sort

- This means that on disk the larger cost is on the movement of the obj (what it is called I/O-bottleneck)

LOWER BOUND We derive the lower bound for permuting in 1-disk case, and then easily extend it to D-disks by dividing by D.

Simple I/O = item does exist either on disk or in memory (no duplicates are created). I/Os can be made simple, cancelling the ~~item~~ transfer if it is not needed, or moving it simple if needed.

Goal: Bound the number t of I/Os needed to generate potentially $N!$ permutations of N obj.
Actually, we distinguish between $t_I + t_O = t$ I/Os (simple)

Let P_i = #permutations potentially generated by (i) I/Os

- $P_0 = 1$;
- P_{i+1} can be expressed from P_i depending on the I/O-op namely if it is INPUT or OUTPUT I/O.

$\boxed{\text{NOTES}}$

- items not necessarily contiguous (some NIL) < int mem / disk

$\boxed{\text{OUTPUT}}$ $\leq \frac{N}{B} + o - 1$ non-empty blocks before o-th operation

#places the output block can go

$$P_{i+1} = P_i \times \left(\frac{N}{B} + o\right) \underset{\substack{\uparrow \\ \text{brutally} \\ \text{since } o \leq N\log_2 N}}{\leq} (N \log_2 N)\, P_i$$

$\boxed{\text{INPUT}}$ Two cases:

1- items together in memory before (block was the output of a previous I/O) $\Rightarrow$ fetched from $\frac{N}{B} + o$ blocks

2- block never fetched in memory before

↳ items never together in memory $\Rightarrow$ $\frac{N}{B}$ blocks

(2) has a factor $b!$ more

(1) $\binom{M-b}{b} \leq \binom{M}{b}$ way of mixing fetched items with items in memory

$$P_{i+1} = P_i \times N \log N \times \binom{M}{B} \quad \text{case 1}$$
$$= P_i \times \frac{N}{B} \times \binom{M}{B} \quad \text{case 2}$$

Actually, the pages are not necessarily always full, so it may be the case that less than B items are fetched in an I/O $\Rightarrow \binom{M}{b_i}$

Calculations are more sophisticated but the main idea is that

$$\underset{\substack{\uparrow \\ t \geq I}}{(N \cdot \log N)^t} \times \underset{\substack{\uparrow \\ t \geq 0}}{\binom{M}{B}^t} \times (B!)^{N/B} \geq N!$$

$$\Rightarrow N \cdot \frac{\log N/B}{B \log M/B + \log N} < \begin{array}{l} \text{sort bound} \\ N \quad \text{if } \log N \geq B \log M/B \quad \boxed{\text{NEVER}} \end{array}$$

$\boxed{\text{MULTI-DISKS}}$: guarantee to fetch D blocks at every I/O

- simplest approach is to look at all disks as one logical disk with page size $B' = DB$.

$\Rightarrow$ Problem $\log_{M/B'} \Rightarrow \log_{\frac{M}{DB}} \frac{N}{M} > \log_{\frac{M}{B}} \frac{N}{M}$

and the larger is D the worse is the approach

$\boxed{\text{RANDOMISED DISTRIBUTION}}$

1. Scan
2. Distribute (D blocks in output buffer)
3. write output buffer

- if Scan is executed on a file whose blocks are equally spread among the D disks, then Scan takes $\frac{S}{DB}$ I/Os to fetch those S blocks.

considerare loading in, not out

- Hence ② must ensure this balanced distribution.

$\boxed{\text{DIFFICULTY}}$ is that the D blocks to write ~~...~~ in ③ may belong to different runs (under ~~...~~ distrib formation), ~~...~~ these are arbitrarily allocated on the D disks, and we wish to pay $O(1)$ for ③

So using "striping" on all runs might generate CONFLICTS in WRITE.

balance if we throw $\#\, D \lg D$ blocks in the run.

$\boxed{\text{SOLUTION}}$ Let N large, ~~...~~ $= \Omega(D \lg D)$ ~~...~~

so every run under formation contains at least $\Omega(D \lg D)$ blocks

- Each I/O-write sends its D blocks according to a random perm of $\{1, \ldots, D\}$
- Every run distributes randomly its b blocks over D disks (aka occupancy problem) $\longrightarrow$ if $b = \Omega(D \lg D) \longrightarrow O(b/D)$ block per disk w.h.p.
- So we have an EVEN distribution, which is perfect to guarantee full throughput for the next ~~...~~ ①

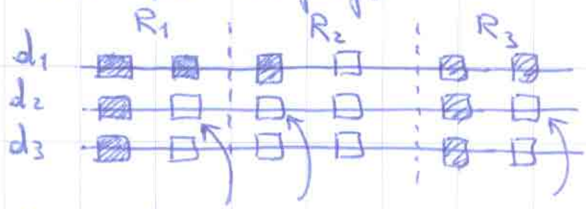
Take blocks of a run which go to the same disk (they are of different I/O-writes) (they are randomly looking at that disk, so also occupancy problem ⇒

$\boxed{\text{NOTE}}$ $b = \Omega(D \lg D)$ hence the assumptions ~~...~~ above

since $\#\text{splitter} = \min\left\{\frac{M}{DB}, \frac{N}{M}\right\}$ N

Greed Sort | No randomization. (SKETCH)

- Problem with merge-scheme: conflict in reading the next D-blocks for merging.



FIG

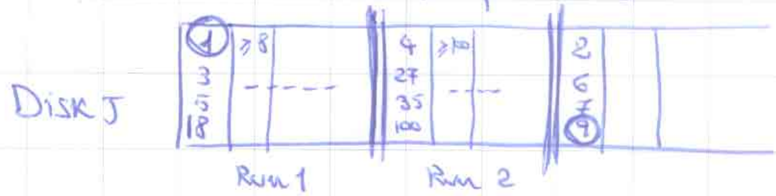
- Prediction sequence ⇒ which block to fetch next $\boxed{\text{EASY}}$
  (sort the min item of every block)

- Given the prediction sequence, we need to fetch D blocks at a time in O(1), they could come from the same disk (notice round-robin helps on one run, but not when we have many of them to be merged).

- Again randomization could help → Randomized cycling is nice, used in output

<u>Greedsort</u> avoids every burden.

- Runs are written in a striped way
- Disks operate "independently", by fetching the best 2 blocks per disk at each I/O-read operation



smallest MIN → m_1
smallest MAX → m_2

- These 2 blocks are merged < smallest B items goes to output; largest B items goes to m_1-run since max of block of m_1 > m_2

- This is an "approximate" merge, provable that every item is $D\sqrt{MB}$ positions from its correct one.

- Typically $D\sqrt{MB} < M$, otherwise use Column sort to fix.