Algorithm Engineering

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction

Teacher: Paolo Ferragina
Exam: written + oral. Midterms in November and December, with exercises.
Classes will be recorded on Microsoft Teams. Also the book "The Magic of Algorithms" is very important: you must be used to talk about these things, not just be able to solve exercises.

Course **Design** and **analysis** of algorithms but also insights about **implementation**, with reference to libraries and considerations about what happens when using certain algorithms. The case of use is **big data**.

0.1.1 Algorithms

Algorithms Knuth's definition: "*a finite, definite, effective procedure that takes some input and returns an output, with the output being the answer to the problem you want to solve.*"

> **Finite**: a **finite sequence of steps**, not only a finite numbers of operations but also **the algorithm must terminate**. `while (true) do ...` will go on forever, so it's not an algorithm. The number of steps must be *very definite and reasonable*, which relates to the efficiency of the algorithm.
>
> **Definite**: the steps are definite in an unambiguous way.
>
> **Effective**: every step is basic, atomic, something that we can execute in small or constant time, constant is not a very precise word (seconds? Milliseconds?) so we will accept the "small time" rough definition. Also, the mapping input $\rightarrow$ output must always be correct, which is the biggest difference with IA. An algorithm outputs the correct output for each input.

RAM Random Access Machine, CPU $\leftrightarrow$ M, classical computing model (Von Neumann machine), the memory can read any place in constant time.
We will make a more sophisticated step. But without presenting very complicated models. We need a good balance, not a perfect but a *better* approximation than the RAM.

Analysis Let's take an algorithm A and let's find a function $T_A(n)$ that describes the time complexity of A. n is the input size, the number of items that the algorithm has to process. The time that A will take will be in hours, seconds or milliseconds based on the machine, but we approximate that time taken with the number of steps that are computed. Also, the number of steps depends not only on the number of items but on the items themselves, too. So we usually analyze the worst case scenario, or less often the average scenario. By analyzing the worst case scenario we can figure out the worst or "maximum" number of steps. **Asymptotic analysis**.
We want to exploit the characteristics of the various types of memory.

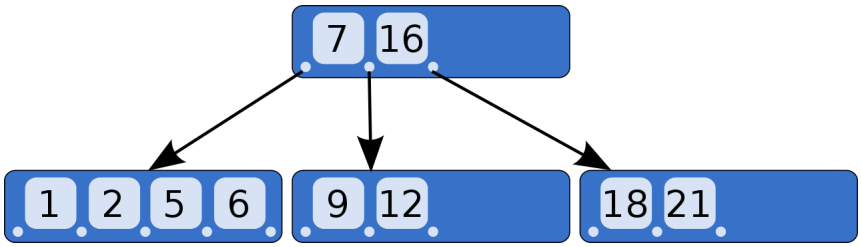
We will count not all the steps but the I/O ops, with a **2-level memory model**: the **first level** is the **internal fast memory** (cache + RAM) and the **second level** is the **mass unbound memory** (disk). In small memory situations, the first level can be interpreted as cache and the second level as internal memory, other times the first level is the internal memory and the second level is unbound slow memory.

> **Spatial Locality**: access near items
>
> **Temporal Locality**, or small working set: far apart items used often so we can exploit their presence in the cache

Poly vs Exp time complexity Let's say we have three algorithms with n, n^2 and 2^n in time complexity respectively. Let's express the time complexity fixing t time and counting how many items we can process in t time. In the first case is linear, so $n = t$, the second is $n = \sqrt{t}$ and the third is $n = \log_2 t$. If we have a k times faster machine, we can imagine that we are using the original machine for k more times, or k original machines in parallel. So in this case $n = kt, n = \sqrt{kt}$ and $n = \log_s kt$. The linear algorithm takes full advantage of the k times faster machine, the second algorithm has a small advantage of a multiplication factor $\sqrt{k}$ and the last a negligible advantage of a sum factor of $\log_2 k$, which is basically none.

Example $A[1,n]$ integer array of which we want to compute the sum. The number of items, then, is n.
First situation: first we load B items and process them, then the next B items and so on... $\Rightarrow$ # I/O $= n/B$, which we will see often and is called the **scan cost**, because we need to see each element, in batches of B elements. So B **is the size of the memory page**.
But we can follow a different approach: we take the first item of each batch of B elements, then the second items and so on, which will take n steps, but this will be possibly slower because this method takes more I/Os ops, n I/O ops. The larger the jumps the more I/O ops we do. The model doesn't distinguish between local and random I/Os.

Binary Search Array of n elements. We pick the middle element, we go to left/right, middle element of the section and so on. The time complexity is $T(n) = O(\log_2 n)$, but we have a lot of I/Os and big jumps, so elements in different and far pages. We have $log_2 \frac{n}{B}$, because at a certain point the sub array where we search will be smaller than a page, so we have $\log_2 n$ steps $-\log_2 B$ the last steps inside the page, and for the properties of the logarithms we have $\log_2 \frac{n}{B}$.
So the larger the page the smaller the number of I/Os. One consideration: n is the number of items, B is in kilobytes. So if we consider integers of 8 bytes, we have $\frac{B}{size}$ items, and with $B = 32$ KB $= 2^{15}$ KB we have circa 4000 times, or $2^{15}/2^3 = 2^{12}$.
How can we improve the search? We can consider the B^+-trees. We split the array into the page size B and the array is sorted in ascendant order. The splits are called leaves, and each leaf has a key (one of the elements). We have a page with each key and the next element is the pointer to its page. Above one level, a page with a key of the first key list and a pointer to the key list, a key of the second and so on.
Fetch a page, binary search and follow the pointer. Number of I/Os is $\log_B \frac{n}{B}$ and the number of steps is a binary search for every page, so $(\log_B \frac{n}{B})$ pages $\cdot \log_2 B$



Analysis $n = (1+\epsilon)M$ with $\epsilon > 0$ data outside of the memory. So M is totally full and ϵM is stored in the unbound memory.
The first point is we want to find P(accessing the disk), with a totally random algorithm, is $= \frac{\epsilon M}{n} = \frac{\epsilon \cancel{M}}{(1+\epsilon)\cancel{M}} = P(\epsilon)$

The second point is the average time of a step. $\sum_x P(X = x) \cdot x$ with X the variable of which we compute the average with that formula. X is time, in this case. The time is 1 in case of computing, and varies in case of accessing the memory. So we have the multiply the cost of access times the probability (of computing, going internal, going to the disk). Internal memory access costs 1, while on disk is larger and we say that the cost is c. So the average, with let's say a probability of memory access, $= (1-a) \cdot 1 + a(P(\epsilon) \cdot c + (1 - P(\epsilon)) \cdot 1)$ but $0 < a < 1$ and $1 - P(\epsilon)$ are very small, so we can rewrite as $= a \cdot P(\epsilon) \cdot c = a \cdot \frac{\epsilon}{1+\epsilon} \cdot c + O(1)$. If $a = 0$ then no memory access so the cost is constant. The larger is a the more memory access, the more the term is important, which is exactly what we want to capture. Usually, $a = 0.3 = 30\%$ and $c = 10^6$ the gap between accessing the disk and accessing the internal memory.
So $\frac{\epsilon}{1+\epsilon} \cdot 0.3 \cdot 10^6 = \frac{\epsilon}{1+\epsilon} \cdot 300000$. If $P(\epsilon) = 0.001$ the avg time of a step is $0.001 \cdot 300000 = 300$, so the disk has a lot of impact even with only a thousandth of memory access being on disk: the avg cost is 300 and not 1.

Sorting and permuting Given an array $S[1,m]$ and a permutation π, the permutation problem asks to permute S according to π. For example $S = [A,B,C,D], \pi = [3,1,2,4]$ then π tells that the $\pi[0] = 3$ item goes to the first position. So $S_\pi = [C,A,B,D]$.

```
for i = 1 to n:
        S'[i]=S[pi[i]]
```

Which costs $\Theta(n)$

	PERM	SORT
RAM	n	$n \log n$
2-level memory	$\min\{n, C_{sort}\}$	C_{sort}

Solving the permuting in a scan + sort kind of way, -altrimenti- we have to do a disk access per value.

scan S : $\langle$S[i], i$\rangle$ which is $\langle$item, position$\rangle$
Which creates $\langle$A, 1$\rangle$ $\langle$B, 2$\rangle$ $\langle$C, 3$\rangle$ $\langle$D, 4$\rangle$
This costs $O(\frac{n}{B})$

scan π : $\langle\pi$[i], i$\rangle$ which is $\langle$src, dst$\rangle$
Which creates $\langle$3, 1$\rangle$ $\langle$1, 2$\rangle$ $\langle$2, 3$\rangle$ $\langle$4, 4$\rangle$
This costs $O(\frac{n}{B})$

sort by first component of the sequence $\langle\pi$[i], i$\rangle$
Which creates $\langle$1, 2$\rangle$ $\langle$2, 3$\rangle$ $\langle$3, 1$\rangle$ $\langle$4, 4$\rangle$
We take the item in source and move to destination

parallel scan $\langle$A, 1$\rangle$ $\langle$B, 2$\rangle$ $\langle$C, 3$\rangle$ $\langle$D, 4$\rangle$
$\langle$1, 2$\rangle$ $\langle$2, 3$\rangle$ $\langle$3, 1$\rangle$ $\langle$4, 4$\rangle$
Which creates $\langle$A, 2$\rangle$ $\langle$B, 3$\rangle$ $\langle$C, 1$\rangle$ $\langle$D, 4$\rangle$

sort by second component
Which creates $\langle$C, 1$\rangle$ $\langle$A, 2$\rangle$ $\langle$B, 3$\rangle$ $\langle$D, 4$\rangle$

scan
Which creates [C, A, B, B]
This costs $O(\frac{n}{B})$

The I/O cost is 4 scan + 2 sort = $O(\frac{n}{B}) + 2 \cdot O(C_{sort})$ sorting cannot cost less than $\frac{n}{B}$ because we can't improve that, so $O(C_{sort})$

So we have proposed upper-bounds = algorithms for the sorting and the permuting problem. The permuting problem can be solved in $O(\frac{n}{B})$+ sorting I/Os. Moving n items takes $\Theta(n)$ I/Os.
Sorting n items in a two level memory of size M for internal memory and B for the disk page size, costs $O(\frac{n}{B} \cdot \log_{\frac{M}{B}} \frac{n}{M})$ with $L = \log_{\frac{M}{B}} \frac{n}{M}$, and often written as $\overline{O}(\frac{n}{B})$ with the overline or over tilde that means that is a scan.
L consists of the base $\frac{M}{B}$ and the argument $\frac{n}{B}$ which means that with a larger memory I'd like it to be faster, with bigger M the argument decreases and the base increases so the logarithm shrinks a lot. With a bigger B page size, $\frac{n}{B}$ decreases but the base increases.
$\frac{M}{B}$ = how many pages I can keep in the internal memory. Let's say $n = 2^{40}$, $M = 8\text{Gb} = 2^{33}$ and $B = 32\text{Kb} = 2^{15}$, then

$$\log_{\frac{n}{B}} \frac{n}{M} = \frac{\log_2 \frac{n}{M}}{\log_2 \frac{M}{B}} = \frac{\log_2 \frac{2^{40}}{2^{33}}}{\log_2 \frac{2^{33}}{2^{15}}} = \frac{\log_2 2^7}{\log_2 2^{18}} = \frac{7}{18} < 1$$

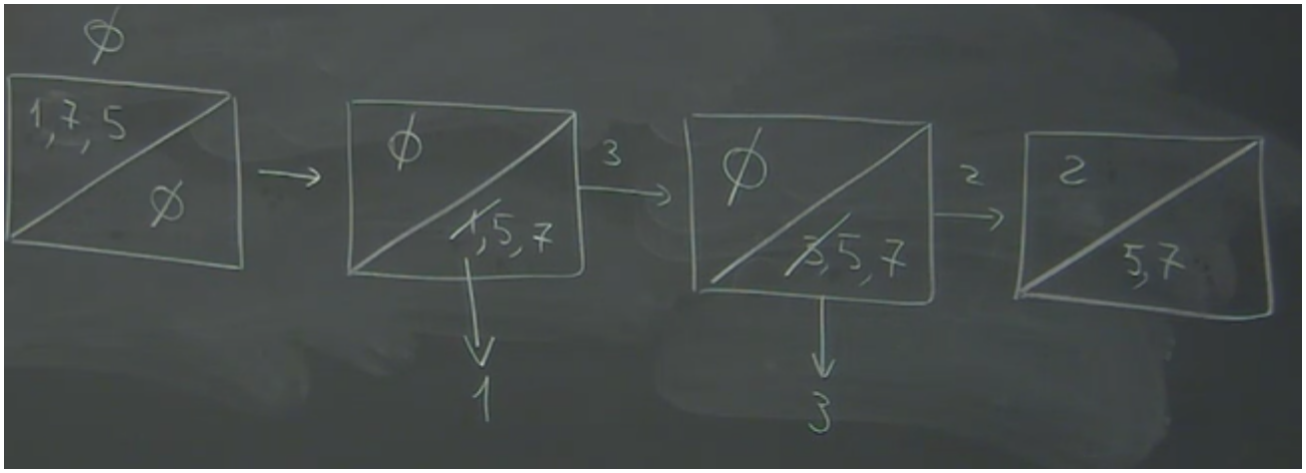
Let's see when is sorting preferred to moving items or viceversa

$$\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M} < n \Leftrightarrow \log_{\frac{M}{B}} \frac{n}{M} < B$$

We have $B = 1$ in the RAM model, $B = 32\text{Kb}$ in the 2-level model compared to $\log_{\frac{M}{B}} \frac{n}{M} = 2$ or 3. In practical situations with disk, sorting is better than moving numbers. If $B = 1$, in the RAM model, $M = O(1)$, then sorting is worse than moving.

Sorting Let's consider binary merge sort, which in the worst case costs $O(n \cdot \log_2 n)$. Let's evaluate the I/Os in the case that $n >> M$ so the array can't be stored entirely in internal memory. Since it's based on the merge procedure, we can evaluate the cost of the merge and multiply by the number of levels. It loads a page every time it needs it and writes a pages every time it fills one. So if the two arrays to merge are long l, it makes $\frac{l}{B}$ I/Os. So it takes $O(\frac{n}{B} \log_2 n)$ I/Os.
Sorting = computing the sorted permutation + implement the sorted permutation. So we can say that sorting $\geq$ permuting, is at least difficult as.... In the RAM model the $\geq$ is $>$, strict, because sorting is $n \cdot \log n$ and computing is the real issue, because implement is linear. In the 2-level memory model they are almost equivalent. This considering **atomic items**, integers o non-splittable strings.
The mismatch is the base of the logarithm. The binary merge sort is $O(\frac{n}{B} \log_2 \frac{n}{M})$: partitioning the array in blocks of size M, the size of the memory, called runs. Can we generate runs longer than M in few I/Os? On average we will be able to create runs of size $2M$, which saves 1 full scan of data (which, for large data, is a lot of time saved).

Snow plow algorithm Sort item that you can, leave item that you cannot.
S = 1, 7, 5, 3, 2, with M = 3 items. The memory is divided in two parts: a min heap, items that are still unsorted, and an unsorted part, the snow that we cannot clean. Start from the memory with everything unsorted: min heap empty and only unsorted items. We put M unsorted items, so 1, 7, 5 in the unsorted part. We sort the unsorted items and move to the min heap, with the unsorted part left empty. We pick the minimum element and write it out of the memory. We have emptied a position, and we can fetch another item in the unsorted part, the 3. The new item is compared to the minimum. 3 is larger than the current minimum, so is written inside the min heap, which is now 3, 5, 7. We write again out the minimum, 3, and fetch another item. 2 is smaller than 3, so it goes in the unsorted part. At some point the min heap will be empty, the unsorted part will be full and we restart the phase. We pay 1 I/O as soon as we write out B items.



Let's prove that the runs are 2M on average. We start with $|U| = M$ and $|H| = 0$, with U unsorted part and H heap part. Continuing we read items, with $T = \#$ items read. The phase has processed $T + M$ items, T read and M already in memory. At the end of the phase, the min heap is empty $|H| = 0$ and $|U| = M$ and T items are written out. So T is the length of the run, we have to compute it, by making hypothesis about the distribution of the items, the probabilities of going to U and H. P(item read goes to U) = 1/2, a totally random situation. By changing the probability we change how much sorted is the sequence. The more sorted is the sequence, the smaller the probability of going to U. $\text{avg}(|U|) = E[|U|] = T/2$. Given that $|U| = M$, then on average $E[|U|] = \frac{T}{2} = M$ so average $T = 2M$

Exercise $M = 2$ and $S = 1, 8, 3, 2, 5, 0, 4, 6 \ldots$

Issues Binary Mergesort doesn't always exploit all the memory. Because after creating the M-long first runs, by merging we need $3B$ for reading the memory (one for the first run, one on the second run and one page on the output), so $3B << M$, a lot of unused memory. We could fetch 2 pages per run, but the second page can be used only after the first page, so no advantage in allocating all data. Since there's a sequence of processing, even if a load immediately all the pages, we do not have much advantage in doing so. So we would like to merge k runs instead of two runs at a time. Since we want to fill the memory, we want $(k+1)B = M$, k pages for k runs plus an output page. So $k = \frac{M}{B} - 1$ which we can approximate with $\frac{M}{B}$. Each run generates l/B I/Os, and the merged run is long kl elements.
So every run is of M elements which take $\frac{M}{B}$ I/Os, for n/M runs. So the total cost is $O(n/M \cdot \frac{M}{B}) = O(\frac{n}{B})$ for creating runs. ?So $\log_k n/M$?.
The cost of mergesort is $O(\frac{n}{B} + \frac{n}{B} \cdot \log_k n/M)$ which we have seen with $k = \frac{M}{B}$.
With improve B, M by compression. So 3, 5, 10 is written as 3 (the first item) and storing the gaps (gap encoding) so 2, 5,... with variable length encoders.

Let's rewrite the bound

$$\frac{n}{B} \cdot \log_{\frac{M}{B}} \frac{n}{M}$$

We don't know how items are consumed on the disks, given D disks. With $k = 2, D = 2$ we can take advantage of the parallelism on the first load. But once loaded in memory, we may consume the pages asymmetrically, so we may end up reading a lot from a disk and very few times from the other. Possibly, every page on the first disk is used before the second page on the second disk. So by adding more disks we take full advantage because they are the denominators. The know optimal bound is:

$$\Theta\left(\frac{n}{BD} \cdot \log_{\frac{M}{B}} \frac{n}{BD}\right)$$

For 1 disk is $O(\frac{n}{B} \cdot \log_{\frac{M}{B}} \frac{n}{B})$
For D disks we can consider one big disk with $B' = D \cdot B$ page size, so the heads of the disks are fixed together and move together. We do not change the algorithm, so the complexity is simply $O(\frac{n}{B'} \cdot \log_{\frac{M}{B'}} \frac{n}{B}) = O(\frac{n}{DB} \cdot \log_{\frac{M}{DB}} \frac{n}{DB})$ but with D in the base of the logarithm makes the value larger, but the impact is small and the simplicity of the method makes it worth it. This is called **disk striping**.

Binary Decision Trees Every node a comparison left branch $a < b$, right $a > b$ and it represents an algorithm. The leaves there are permutations, total number of leaves is $n!$. The question is how big should be the tree so that it's a sorting algorithm, it grows by 2^t nodes for level t, so $2^t > n!$ and we have the lower bound.

Lower bounds About sorting
There are also lower bounds about permuting, an extra for the oral.
Binary decision trees proved lower bound of sorting in the RAM model. Now let's see the I/O lower bound, so not counting comparisons but read/written pages. The main idea is that whenever i fetch a page I fetch B items, not just two, so many comparisons. And the page goes in internal memory where there are more items which we compare. Whenever the items goes to internal memory we have to avoid repeating comparisons.
The memory M consists of several items, we bring the page of size B in internal memory and we have $M - B$ other items in the memory. One I/O executes many comparisons. If $B = 1$ and $M - B = 3$, so 3 items in M and 1 item brought from disk. How many comparisons? We can assume that the items in memory are sorted, so the item brought can go in 4 position (before the first, between the first two...). In general, the B items can enter the $M - B$ items in n ways. The memory consists of M cells, so $n = \binom{M}{B} \cdot B!$ with $B!$ because we have to account for the shuffling.
So two situations

1. A new read, a read of an input page for the first time
   # new reads $= \frac{n}{B}$
2. An old read, all other read-cases
   $t - \frac{n}{B}$

t I/Os, can be a new or an old read. Either one of the $\frac{n}{B}$ pages for the first time or any other page (or one of those pages for the second/third/... time)
Two kinds of nodes, one node refers to a new I/O the other to an old I/O.
In a new I/O we have $\binom{M}{B} \cdot B!$ options, we have to account for the shuffling, in the second case we have $\binom{M}{B}$ options.
So in any path $\frac{n}{B}$ are new and $t - \frac{n}{B}$ are old. So how many steps or levels t are needed to guarantee that the number of leaves is $\geq n!$? Every node is no longer binary and can be one of two kinds. Still a decision tree. So in every step we have $\frac{n}{B}$ new I/Os and $t - \frac{n}{B}$ old I/Os.
permutations that the algorithm can distinguish is

$$= \left(\binom{M}{B} \cdot B!\right)^{\frac{n}{B}} \cdot \binom{M}{B}^{t - \frac{n}{B}} = \binom{M}{B}^{t} \cdot (B!)^{\frac{n}{B}} \geq n!$$

To compute, consider that $\log \binom{M}{B} = b \log_2 \frac{a}{b}$ and solving for t gives the multiway merge sort bound. $t = \Omega(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$

Two sorting paradigms The following:

merge-based paradigm: partition (computationally free, $O(1)$), recursion, recombination (merge, $O(\frac{n}{B})$)

distribution/partitioning-based paradigm: partition (costly part, $O(n)$), recursion, recombination (doesn't exist)

How can we design a multi way quick sort? We can't take one pivot, because that'll be binary. We can take many pivot but we have to guarantee a good usage of internal memory, but more importantly we need that the partition have to balanced: with k pivot I'd like that every partition has $\frac{n}{k}$ items. This requires a theorem.
Let's refresh the quick sort.

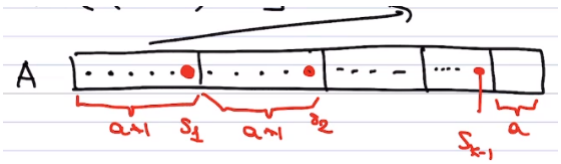
Time complexity of quick sort is $O(n \log n)$ By introducing a random variable $X_{u,v} = \begin{cases} 1 & if S[u] \ compared \ to S[v] \\ 0 & else \end{cases}$ and we want $E[\sum_{u=1}^{m} \sum_{v>u}^{m} X_{u,v} = \sum_u \sum_v P(S[u] \text{ compares to } S[v])$, let's focus on the probability that taken two items they are compared. Comparison in quick sort is particular, may only occur in the partition part. With just one pivot and two parts, then S[u] is compared to S[v] only when one of them is the pivot. So $P(S[u]$ compares to $S[v]) = P(S[u]$ or $S[v]$ is choosen as pivot in one call that involves both of them) Case a , case b no comparison because they'll go to different partitions. Case c is that either u' or v' is taken as pivot, so there's the comparison.
$T(n) = O(n) + [T(n_<)$ or $T(n_>)]$ choice of going to the left or to the right, probability. Let's analyze that probabilities.

3-way partitioning Made with 2 pivot taken at random: it reduces the number of comparisons and it increases the number of swaps. This seems totally ineffective (swaps impacts on memory and comparison on CPU), but the problem here is mispredictions on the CPU, modern CPU do look-ahead and if it badly predicts the future ifs it has to break the stack and go back. So with fewer comparisons it's better. 10% circa speed-up factor.

Bounded Quick Sort Bounding the recursive calls. Transform the deepness of the tree (the number of recursive calls) from n to $\log n$, but the time complexity remains n^2 in the worst case.
This trick is removing tail recursion. Techinques applied by compilers, we'll do by hand.
Generic call, between positions i and j. If $j - i$ is $> n_0$ (so if is sufficiently small, the array is in cache so we go fast) then we do insertion sort. In the other case, with $\frac{i+j}{2}$ being the middle element, we do the partitioning assuming the the pivot on the left part of the middle and left partition of the pivot is smaller so contains less than half elements. So $i \leq p < \frac{i+j}{2}$, then recurse on the left because it has fewer elements, so we do $i = p + 1$, and do the same. Else, recurse on the right partition $j = p - 1$

Multi Way Quicksort Key idea: array S to be sorted, doesn't fit in internal memory: we want to partition S among k buckets $B_1, B_2, B_3, \ldots, B_k$. Every bucket is associated to a pivot: $k - 1$ pivots, a logical pivot $s_0 = -\infty$ at the beginning, $s_{k+1} = \infty$ at the end and $s_1, s_2, s_3, \ldots, s_{k-1}$ between the buckets.
$B_i = \{S[j] \mid s_{i-1} < S[j] \leq s_i\}$, so the "right" pivot is part of the bucket ($s_i \in B_i$). The goal is having balanced $B_i \forall\, i$, so $|B_i| = \frac{n}{k}$.
The idea is similar to marge sort but in reverse: we assume that in memory M we have k buckets of size B, so we want to guarantee that $k \cdot B \leq M$. We sample at random $(k-1)$ pivots from S (done with a scan $O(\frac{n}{B})$ I/Os). Not easy, sampling at random, we will see. We sort the the pivots $s_1 < s_2 < \ldots < s_{k-1}$, with 0 I/Os because those are all in internal memory. But we cannot assume that the buckets in memory can contain the S's buckets. So we fetch B items from S and distribute the items according to the pivots and do until a bucket is full. When a page is full, we write it as the first page of the corresponding buckets (according to the pivot). This takes $O(\frac{n}{B})$ I/Os.

1. Pick at random $(a+1)k - 1$ samples from S, the pivots $\Rightarrow A$
2. Sort the samples in A
3. $s_i = A[(a+1)i]$
   Until the last block has a items



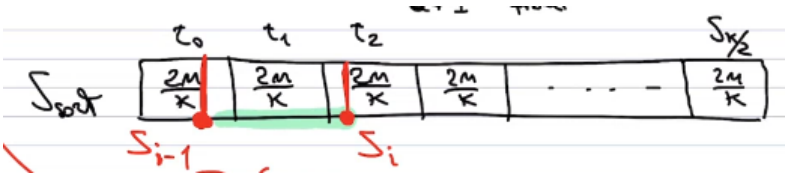
We sampled $(a+1)(k-1) + a = (a+1)k - 1$ items

If $a = 0$ the number of samples is $k - 1$, no oversampling, fast sort ($O(k \log k)$) but no room to play around with the items. If $a > 0$ the sorting costs $O((ak)\log(ak))$. We chose $a = \Theta(\log k)$, only a logarithmic oversampling to get the balance

$$a + 1 = 12 \ln k$$

Balanced $\Rightarrow |B_i| < 4 \cdot \frac{n}{k}$

Proof By contradiction. We want to estimate $P(\exists\, B_i \,||B_i| \geq \frac{4n}{k})$ and prove that it is $\leq \frac{1}{2}$ so that on average two extractions are enough to guarantee that one is correct.
$B_i = \{S[j] \mid s_{i-1} < S[j] \leq S_i\}$ by definition. A_{sort} contains samples from S.
S_{sort} is composed of $\frac{k}{2}$ cells of size $\frac{2n}{k}$: $t_0, t_1, \ldots, t_{\frac{k}{2}}$.
Some pieces of t_i will be totally covered by a bucket B_j, because since B_i has size of at least $\frac{4n}{k}$ it's not possible for it to be totally included into a t_i. B_j has twice the size, so at least one t_i will be totally covered by it. So this means that $P(\exists\, B_i \,||B_i| \geq \frac{4n}{k}) \leq P(\exists\, t_j \,|\, t_j$ is covered entirely by some $B_i)$. So the element before the beginning of B_i, which is s_{i-1}, it's contained in some point of some t_{j-1} and the last element, s_i, is in some part of t_{j+1}.



This means that the a samples are distributed in that part, between s_{i-1} and s_i: part in t_{j-1}, part in t_j and part in the beginning of t_{j+1}. How many samples inside t_j, the block covered by B_i? They are surely $< a+1$, so
$P(\exists\, t_j \mid t_j$ is covered entirely by some $B_i) \leq P(\exists\, t_j \mid t_j$ contains less than $a+1$ samples)
We now apply the **union bound**: the probability of existence is upper bounded by the number of possibilities times the probability of one of the events. So $P(\exists\, t_j \mid t_j$ contains less than $a+1$ samples$) \leq \frac{k}{2} \cdot P(t_1$ contains less than $a+1$ samples). We need to estimate the latter, how many samples will go to t_1. $P($sample occurs in $t_1) = \frac{\frac{2n}{k}}{n} = \frac{2}{k}$, and the average number of samples in t_1, given $X_i = 1$ if the ith item is in t_1, is $X = \#$ of samples in $t_1 = \sum_{i=1}^{(a+1)k-1} X_i$. So $E[X] = E[\sum_i X_i] = \sum_{i=1}^{(a+1)k-1} E[X_1] = ((a+1)k-1)\frac{2}{k} \geq 2(a+1) - 1$ because $k \geq 2$ and $2(a+1) - 1 \geq \frac{3}{2}(a+1)$.

$P(p \in D) =$ positive events / total number $= \frac{|D|}{n} < \frac{m}{n} \leq \frac{\frac{n}{2}}{n} = 1$
n is known, streaming model, $M = 2$, $n = 8$ items which are $[a, b, c, d, e, f, g, h]$, $P = [.5, .5, 0, .5, 1, 0, 1, 1] \leftarrow Rand(0, 1)$
$P($sample $i_j) = \frac{m-s}{n-k+1}$ with s being

$s = 0, j = 1$ extracts a probability (.5) and compare P against the $P($sample $i_j)$
$P = .5 \leq \frac{2-0}{8-1+1} = \frac{1}{4}$, first item not picked

$s = 0, j = 2, P = .5 \leq \frac{2-0}{8-2+1} = \frac{2}{7}$, not picked

$s = 0, j = 3, P = 0 \leq \frac{2-0}{8-3+1} = \frac{1}{3}$, c is picked

$s = 1, j = 4, P = .5 \leq \frac{2-1}{8-4+1} = \frac{1}{5}$, not picked

$s = 1, j = 5, P = 1 \leq \frac{2-1}{8-5+1} = \frac{1}{4}$, not picked

$s = 1, j = 6, P = 0 \leq \frac{2-1}{8-6+1} = \frac{1}{3}$, f is picked

$M = 2$ and we've picked two items, the end

In the reservoir sampling, still streaming model but n is unknown. $S = [a, b, c, d, e, f, g, h, i \ldots]$, $M = 3$. Reservoir initialized with first M items, $R = [a, b, c]$. Each value in S has an integer $\_, \_, \_, 2, 4, 1, 2, 3, 1$, with the last 1 is the h position, always drawn from the interval $[1, j]$. The number assigned are always smaller than the position j.

We consider the item d, extracts position $2 < 3 = M$ and substitutes b in the reservoir, $R = [a, d, c]$

e, position $4 > 3$ nothing changes

f, position $1 < 3$ and substitutes a, $R = [f, d, c]$

g, $R = [f, g, c]$

h, $R = [f, g, h]$

i, $R = [i, g, h]$

Disk Striping $D > 1$. Several disks, the pages of the disks are linked: first page of all D disks are a single page. So it's considered as a single disk with $B' = D \cdot B$
In sorting, the bound is $O(\frac{n}{DB} \cdot \log_{\frac{M}{DB}} \frac{n}{M})$ over D disks with disk striping. The lower bound is $\Omega(\frac{n}{DB} \cdot \log_{\frac{M}{B}} \frac{n}{M})$

$$\frac{\frac{n}{DB} \cdot \log_{\frac{M}{DB}} \frac{n}{M}}{\frac{n}{DB} \cdot \log_{\frac{M}{B}} \frac{n}{M}} \geq 1$$

$$\frac{\log_2 \frac{n}{M}}{\log_2 \frac{M}{DB}} \cdot \frac{\log_2 \frac{M}{B}}{\log_2 \frac{n}{M}}$$

$$\frac{\log_2 \frac{M}{B}}{\log_2 \frac{M}{B} - \log_2 D}$$

$$\frac{1}{1 - \frac{\log_2 D}{\log_2 \frac{M}{B}}}$$

$$\frac{1}{1 - \log_{\frac{M}{B}} D}$$

$\frac{M}{B}$ tipically thousands, D at most 10-20, so $\log_{\frac{M}{B}} D < 1$. $M \to \infty \Rightarrow 1$ optimal. $M \to DB \Rightarrow \infty$, very very slow.

Randomized data structures

TREAP From binary search TREe + hEAP.
Every node is two part: a part is the key and the other is the priority. Binary search tree according to the key and a heap according to the priority. To distinguish, letters for the key and numbers for the priority and max heap.
Left part the keys will be smaller than the root, the right will be larger. The left and right tree, being a maximum heap, have smaller priorities. So the root has the maximum priority.
Let's consider the following TREAP:
3-sided range query