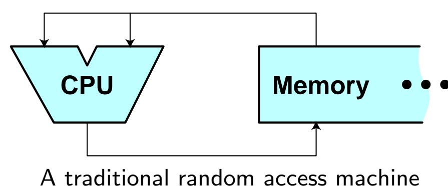
How many Libraries of Congress per second can **your** *server handle?*
— Avery Brooks, IBM television commercial, 2001

1 Introduction

Computer scientists traditionally analyze the running time of an algorithm by counting the number of executed instructions; this is usually expressed as a function of the input size n. This type of analysis implicitly assumes that every instruction takes the same amount of time, at least up to small constant factors. In particular, the standard Random Access Machine (RAM) model of computation assumes a machine with an unbounded amount of available memory, and that any desired memory location can be accessed in unit time.

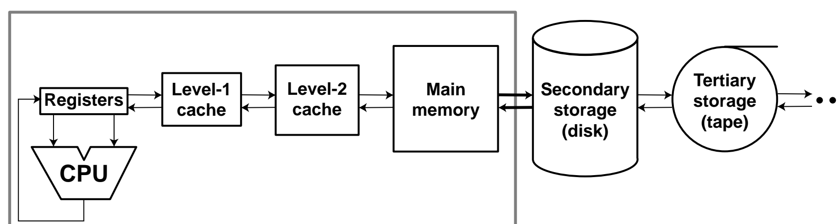


A traditional random access machine

This model predicts real-world behavior fairly well when the amount of memory that the algorithm needs to use is smaller than the amount of memory in the computer running the code. If the code needs to use more memory than the computer has, most operating systems compensate by storing part of the memory image on disk, to be retrieved later if necessary. In this situation, we are faced with two different kinds of memory, with access times that differ by several orders of magnitude: about 10 milliseconds versus about 100 nanoseconds for typical early 21st century computers.[1] Moreover, even though the speeds of memory and disks are both increasing, memory speed is improving much more rapidly. Thus, for algorithms that use lots of data, *disk usage* is a much more appropriate measure of running time than CPU cycles.

This slowdown is a real concern for an increasing number of real-world problems, where even the input is too large to fit into main memory. Example application domains include web search engines, VLSI verification, computer graphics, geographic information systems (GIS), and databases maintaining astronomical, geological, meteorological, or financial data. In such applications, we may be faced with terabytes or even *petabytes* of input data.[2]

Most modern computer systems have several levels of memory, each level larger and slower than the one before it—registers, level-1 (on-chip) cache, level-2 cache, level-3 cache, main memory, disk, tape, tape library, the Google cache, the world-wide web, etc.—but by far the biggest performance hit is at the memory/disk interface.



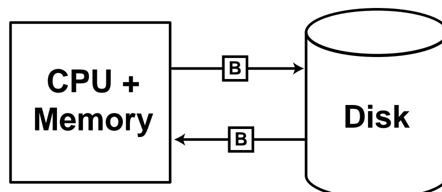
A more accurate (but still oversimplified) model of a modern computer system.

[1]Truly die-hard theory people might respond say "Who cares? $100,000$ is still only a constant factor!", just before they're tarred, feathered, and run out of town.

[2]A petabyte is $2^{10} = 1024$ terabytes, 2^{20} gigabytes, or $2^{50} \approx 10^{15}$ bytes. The Library of Congress contains between 10 and 20 terabytes of (raw ASCII) text. Combining the text in all U.S. academic research libraries would give us about 2 petabytes. Less than 500 petabytes (half an exabyte) of raw text has *ever* been written or printed.

Most of the time spent accessing a disk is spent moving the read/write head to the desired track and waiting for the desired sector to rotate underneath it; once everything is in place, reading or writing a contiguous block of data is just a matter of letting the disk continue to rotate. Thus, to partially offset the high cost of accessing the disk, data is usually read from and written to disk in relatively large contiguous *blocks*.

With all the pieces in place, we can now describe the *standard external-memory model of computation*, first established by Alok Aggarwal and Jeff Vitter in 1988:[3]



The external-memory machine.

- The computer consists of a single processor, an *internal memory* that can hold M objects, and an unbounded *external memory*.
- In a single *input/output* operation, or *I/O*, the computer can transfer a contiguous block of B objects from internal memory or external memory or vice versa. Clearly, $1 \leq B \leq M$.[4]
- The *running time* of an algorithm is defined to be the number of I/Os performed during its execution.[5] Computation that uses only the data in internal memory is absolutely free.
- The *size* of a data structure is defined to be the number of blocks that comprise it.[6]

For notational convenience, whenever we use an upper-case letter to denote a certain number of *objects*, the corresponding lower-case letter will denote the corresponding number of *blocks*. Thus, $m = M/B$ is the number of blocks that will fit into internal memory. We will consistently use N to denote the number of input objects for a problem, and $n = N/B$. We will always assume that $N > M$, since otherwise, the model would allow us to do everything for free! For problems that produce large outputs, we use K to denote the number of output objects, and $k = K/B$ to denote the (ideal) number of output blocks. Most of the time, we will implicitly assume without comment that the lower-case variables are integers.

B = number of objects per block	
M = number of objects that can fit into main memory	$m = M/B$
N = number of input objects	$n = N/B$
K = number of output objects	$k = K/B$

Common parameters for analyzing external-memory algorithms and data structures.

1.1 Searching

Let's start with a simple example. Given a file of N objects, stored in a contiguous sequence of $n = N/B$ blocks on disk, how do we determine whether another *query object* x is in the file? With no other assumptions about the problem, the following trivial algorithm does the trick:

[3] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM* 31(9):1116–1127, 1998.

[4] Typical values, assuming 4-byte objects, are $B = 2^{11}$ and $M = 2^{27}$.

[5] This is non-standard; most research papers say "I/Os" instead of "time".

[6] This is non-standard; most research papers say "blocks" instead of "space".

```
ExternalScan(x):
  for i ← 1 to n
      read block i of the file into memory   (*)
      if the block contains x
          return True
  return False
```

Each execution of line $(*)$ contributes one I/O to the running time; everything else is free. Thus, the total running time of this algorithm is $O(n) = O(N/B)$ I/Os. Will will usually refer to this as "linear time".

In the standard internal-memory model, we can improve search time from $O(n)$ to $O(\lg n)$[7] by pre-sorting the data and using binary search. A similar trick applies here. If the N input objects are stored on disk in sorted order, then we can perform binary search in $O(\lg n) = O(\lg(N/B))$ I/Os. Note that the search algorithm effectively stops as soon as we've identified the unique block that might contain x.

1.1.1 B-trees

We can improve the external search time even further using a data structure called a *B-tree*. Since their introduction by Rudolf Bayer and Edward McCreight in 1970,[8] B-trees have become one of the most ubiquitous data structures on earth. B-trees use linear space—$O(n) = O(N/B)$ blocks to store N objects—and support searches, insertions, and deletions in only $O(\log_B n) = O((\log n)/(\log B))$ I/Os in the worst case.

Roughly, a B-tree is an $\Theta(B)$-ary search tree with $\Theta(N/B)$ leaves. Specifically, the root has between 2 and B children; each other internal node has between $B/2$ and B children; and each leaf (assuming there is more than one) stores between $B/2$ and B objects. Note that the data associated with any node in the B-tree fits into a single block.[9] Each internal node stores a sorted sequence of search keys; the number of search keys is one less than the number of children. The r search keys at the root split the data into $r + 1$ subsets of roughly equal size, where the elements of each subset are contiguous in sorted order; the subtrees of the root are B-trees for these subsets. In most popular variants of B-trees, the actual data is stored in the leaves, and the search keys in the internal nodes are artificial 'pivot' values.

To search for a query object x, we load the block of r search keys at the root, find the pair of adjacent search keys that bracket x, and search recursively in the corresponding subtree. At each level of the tree (except possibly the first), the range of possible positions for x drops by a factor of at least $B/2$; the algorithm stops when we identify the unique leaf block that might contain x. Thus, the number of I/Os is $O(\log_B n)$, as promised.

To insert a new object x, we locate the appropriate leaf ℓ using the search algorithm we just described. If ℓ has room, we store x there and return. Otherwise, we *split* ℓ into two leaves ℓ^- and ℓ^+, each storing half the elements of ℓ, and add x to the appropriate new leaf. This split causes the insertion of a new pivot element at the parent of ℓ, which is handled exactly as an insertion into a leaf node. Thus, splits can propagate all the way to the root. Some of this propagation can be limited by *donating* pivot elements in full nodes to non-full siblings. The total time per insertion is $O(\log_B n)$.

[7] lg means $\log_2$. Normally the base of the logarithm doesn't matter in $O()$ bounds, but I want to emphasize here that the base is a *constant*.

[8] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. SIGFIDET Workshop 1970: 107-141. Acta Informatica 1:173-189, 1972.

[9] It will sometimes be useful to have B-trees whose fanout is $\Theta(\sqrt{B})$, or $\Theta(m)$, or some other value besides B.

Deletion is the opposite of insertion. We begin by deleting the target element x from its leaf ℓ. If ℓ now has less than $B/2$ elements, we either *steal* elements from or *fuse* ℓ with one of its immediate siblings. Like splits, fuse operations can propagate all the way up to the root. The total time for a deletion is $O(\log_B n)$.

1.1.2 Lower Bounds

In the internal-memory model, binary search is provably the best possible *comparison-based* search algorithm. More formally, we can define a *comparison tree* as a binary tree, where each internal node stores a pair of indices into the input array, and each leaf stores an integer between 0 and n. To compute with a comparison tree, we compare the two items indicated at the root, branch to the left or right depending on which of the two items is smaller, and continue recursively until we reach a leaf. The value stored at the leaf represents the number of items smaller than the query x. Since there are $n+1$ possible outputs, the decision tree has at least $n+1$ leaves, and thus must have depth at least $\lceil \lg(n+1) \rceil = \Omega(\lg n)$.

We can make a similar argument in the external-memory model, under the assumption that the computer only compares (and possibly reorders) pairs of objects in its internal memory, including the query x. With this assumption, we can model any algorithm as a decision tree, but with much higher degree. At each node in the tree, we load a block of B items from the disk, and then branch according to the rank of x in that block. (We do *not* charge for the internal comparisons required to compute that rank.) As before, the decision tree has at least $N+1$ leaves, and each node has at most $B+1$ children, so the depth of the decision tree must be at least $\lceil \log_{B+1}(N+1) \rceil = \Omega(\log_B n)$.

1.2 Sorting

There are several internal-memory sorting algorithms that run in $\Theta(n \log n)$ time, and this time bound is optimal in the comparison-tree model. One of the many optimal algorithms inserts the items one at a time into a balanced binary tree and then performs an in-order scan of the tree. If we do the same thing in the external-memory setting, we obtain an algorithm to sort N items using $O(N \log_B n)$ I/Os. Surprisingly (until you think about it for a while), this time bound is *far* from optimal.

1.2.1 Lower Bounds

Before we look at algorithms, let's try to establish a lower bound in the external comparison-tree model. As before, we model any algorithm as a high-degree decision tree. At each node, we write a block of data from memory to disk, load a block of data into internal memory, and then branch according to the permutation of the M items in internal memory. (Effectively, we assume that as soon as any block is loaded, the entire internal memory is sorted for free.) Since the decision tree has at least $N!$ leaves, and each node has fanout at most $M!$, this gives us a crude lower bound of $\lceil \log_{M!} N! \rceil = \Omega((N \log N)/(M \log M))$ I/Os.

This bound is obviously not tight; any sorting algorithm must examine all its input data, and that requires $\Omega(n)$ time. We can improve it by observing that not all $M!$ memory permutations are possible at *any* node in the tree. The memory permutation at a node v must be consistent with the memory permutation at the parent of v, and the set of objects in memory at these two nodes differs by at most B objects. Moreover, we can assume without loss of generality that the objects within each block of the input are already sorted before the algorithm begins—this requires only n additional I/Os to enforce— and that the objects within any block we write to disk are also sorted. With these assumptions, the number of possible output permutations is $N!/B!^n$, and the fanout of

any node is at most $\binom{M}{B}$; the memory permutation can be described uniquely by listing the ranks of the B freshly-loaded objects. It follows that the depth of the tree is at least

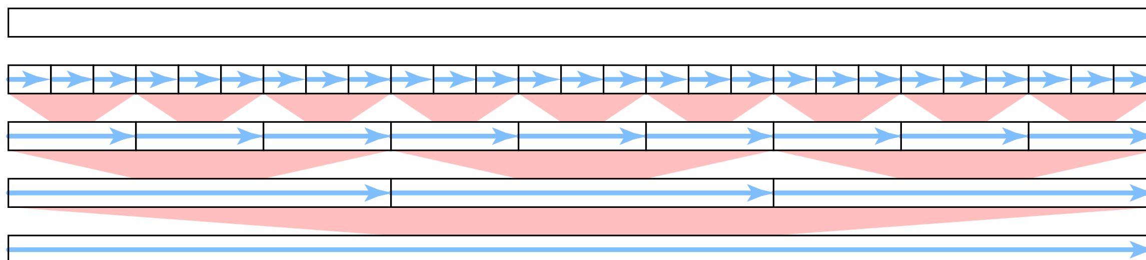
$$\frac{\log(N!/B!^n)}{\log\binom{M}{B}} > \frac{\log(N!/B!^n)}{\log(M^B/B!)} = \Omega\left(\frac{N\log N - n(B\log B)}{B\log M - B\log B}\right) = \Omega\left(\frac{N\log n}{B\log m}\right) = \boxed{\Omega(n\log_m n)}.$$

This is the correct time bound! There are at least three different sorting algorithms that use only $O(n\log_m n)$ I/Os: mergesort, distribution sort (the external equivalent of quicksort), and buffer-tree sort (the external equivalent of binary insertion sort).

1.2.2 External Mergesort

External mergesort is a straightforward generalization of internal mergesort. The algorithm starts by repeatedly filling up memory with M input items, sorting those, and writing them back out to disk. This process divides the input file into $N/M = n/m$ sorted runs, each with M elements, and uses exactly $2n$ I/Os (each input block is read once and written once).

The algorithm then repeatedly merges sets of m runs together, until only a single sorted run containing all the input data remains. To begin an m-way merge, we load the initial block of B elements from each of the m runs. We repeatedly move the smallest item in the m runs to a new output block. Whenever the output block becomes full, we write it to disk; whenever one of the run blocks becomes empty, we read the next block from that run. We easily observe that this process never overflows the internal memory.



External mergesort (with $m = 3$).

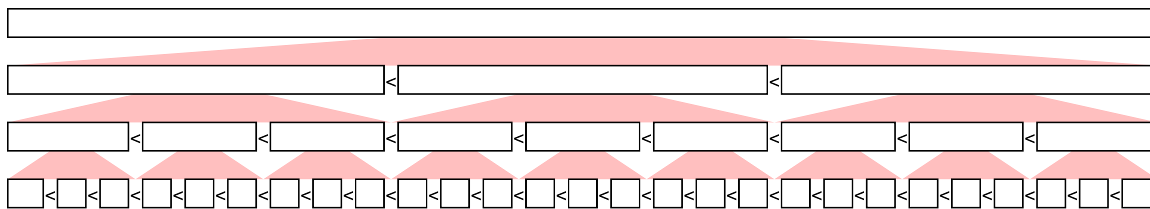
If we always merge runs of equal size, we can partition the execution of the algorithm into phases, where every merge in the ith phase transforms m runs of length $M \cdot m^{i-1}$ into a single run of length $M \cdot m^i$. Each phase reads and writes every input item exactly once and thus requires exactly $2n$ I/Os. In the last phase, we have $M \cdot m^i = N$, so the number of phases is

$$\log_m(N/M) = \log_m(n/m) = \log_m n - 1.$$

Overall, the external mergesort algorithm uses $2n\log_m n = \boxed{O(n\log_m n)}$ I/Os.

1.2.3 Distribution Sort ('External Quicksort')

External distribution sort is the external equivalent of quicksort. Where mergesort partitions the input data trivially and spends all its time carefully merging, distribution sort carefully partitions the data so that merging the recursively sorted partitions is trivial. Intuitively, we partition the input array into m equal-size subarrays so that any elements in an earlier subarray is smaller than any element in a later subarray, and then recursively sort each subarray. The recursion stops when the subarrays fit into single blocks. Provided we can find m 'pivot' elements that nicely partition the array in $O(n)$ I/Os, the resulting algorithm runs in $O(n\log_m n)$ time.



Distribution sort (with $m = 3$).

Unfortunately, finding m equally-spaced pivots is not so simple. An external version of the Blum-Floyd-Pratt-Rivest-Tarjan linear-time median-finding algorithm finds the median element and partitions the array into larger and smaller elements, all in $O(n)$ I/Os. This algorithm can be used recursively (á la quicksort!) to find m equally-spaced elements in $O(n \lg m)$ I/Os. Unfortunately, this approach leads to an overall running time of $O(n \lg m \log_m n) = O(n \lg n)$ I/Os, which is admittedly better than a straightforward internal quicksort, but still worse than optimal.

To make our lives easier, we relax the requirements on the pivot elements in two different ways. First, we require only some number $\mu < m$ of pivots instead of m; the exact value for μ will fall out of the analysis. Second, we only require the pivots to be *approximately* equally spaced. These relaxations will only increase the number of levels of recursion by a small constant factor, so we still obtain an algorithm with running time $O(n \log_m n)$.

To find these relaxed pivots, we follow the "median of medians" approach of the Blum-Floyd-Pratt-Rivest-Tarjan algorithm. We split the N input elements into N/M chunks of size M, and collect every αth element of each chunk into a new array U, for some value of α to be determined shortly. Altogether, U contains N/α elements. This collection process requires only $O(n)$ I/Os. (With no additional cost, we can actually sort the chunks, as we did in the initialization phase of mergesort.) We then use the algorithm described in the previous paragraph to find μ equally-spaced elements in U; these are our pivots. Overall, our pivot-selection algorithm runs in $O((n/\alpha) \lg \mu)$ I/Os and uses $O(n/\alpha)$ blocks; by setting $\boxed{\alpha \geq \lg \mu}$, we guarantee the desired linear running time.

Call an input element *marked* if it is copied to U during the selection algorithm. Each subarray contains exactly $N/\alpha\mu$ marked elements, and between two marked elements in the same chunk of the input array, there are exactly α elements. Thus, each subarray contains *at most* the α elements following each of its marked elements, plus the α elements preceding the first marked element in each chunk, or at most

$$\frac{N}{\mu} + \frac{N\alpha}{M}$$

elements altogether. Similarly, the number of elements in each subarray is at least

$$\frac{N}{\mu} - \frac{N\alpha}{M}.$$

To guarantee that the subarrays are all roughly the same size, we set μ and α so that

$$\frac{N\alpha}{M} < \frac{N}{2\mu} \iff \boxed{\mu \leq \frac{M}{2\alpha}}.$$

Finally, the overall running time of our algorithm is $O(m \log_\mu n)$, so to get the desired optimal time bound, we need $\boxed{\mu \geq m^\varepsilon \text{ for some } \varepsilon > 0}$.

In their 1988 paper, Aggarwal and Vitter used the parameters[10] $\mu = \sqrt{m}$ and $\alpha = \sqrt{m}/4$, but we can obtain a faster algorithm (by a constant factor) by setting $\mu = m/\lg m$ and $\alpha = 2 \lg m$.

[10] They also used a more naïve selection algorithm that finds μ equally-spaced elements in $O(n\mu)$ I/Os, by running the Blum-Floyd-Pratt-Rivest-Tarjan algorithm from scratch μ times. They should have known better; Jeff Vitter was a PhD student of Don Knuth!