Memory Hierarchies

Administrative Information

- Lecturer: Dr. Nodari Sitchinava (`nodari@ira.uka.de`)
- Course grants 5 European Credit Transfer and Accumulation System points
- Two students take notes every lesson
- Oral exam (1-2 month after semester, two dates to choose from)
- Check website for updates: *https://algo2.iti.kit.edu/2053.php*
- 2 homeworks during semester

1 Motivation

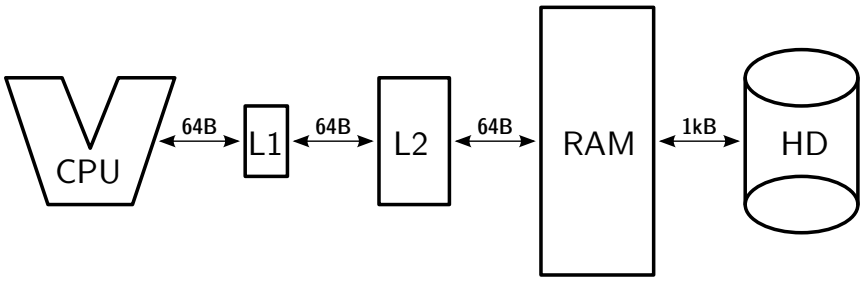
Traditional algorithms courses teach how to design algorithms in the Von Neumann model of computation: data is stored in a single level of memory (RAM) and the CPU can access any item in memory in unit time. This is a major simplification of modern processors which consists of several levels of memory where data might reside and with different access times for each one: from really fast but small caches, to RAM, to slow disks (for data too large to fit in RAM). Even modern caches consist of multiple levels. With the development of multi-core systems in the past decade, the efficient use of memory hierarchy has become even more important when designing parallel algorithms for such systems.

In this course you will learn how to design sequential and parallel algorithms which take advantage of the fast local memories of each CPU on modern (multi-core) architectures.

1.1 Introduction

Is counting the steps of an algorithm enough to derive its runtime behaviour?

In reality, there are multiple hardware layers with different latencies to consider:



type	common size	throughput
L1	64 kB	40-50 GB/s
RAM	2-8 GB	2-10 GB/s
HDD	> 1TB	100 MB/s

Today's multi-core machines have separate cache for each core, so without copying between caches, maximum throughput between CPU and L1-cache could reach up to 400 GB/s in an optimal scenario.

In the worst case, with random accesses to the HDD, throughput can drop down to 1 KB/s due to the harddisk's seek time of roughly 3 to 10 ms.

Therefore, there can be a potential difference as large as 400 GB/s to 1 KB/s between a cache-efficient and a cache-inefficient algorithm.

The difference in access-time can be compared to sharpening a pencil at home and flying to Australia, sharpening it there. Thus, it is increasingly important to carry as many pencils as possible during each trip to reduce the amount of transfers necessary.

Why isn't this taught in the algorithm course?

- Most of the time it works relatively well
- CPU manufacturers design their hardware to accommodate common usage patterns

1.1.1 Cache Locality

Spatial locality occurs when adjacent memory areas are accessed one after another, for instance, when elements of an array are accessed in ascending (or descending) order.

Temporal locality occurs when some element in memory is repeatedly accessed over time.

A cache replacement policy decides which data to keep cached and which to evict from cache.

```
for (i = 0, i < N, i++) {
    sum1 += a[i];
    sum2 += a[i];
}
```

The code snippet above exhibits both spatial and temporal locality:

Spatial because `a[i]` is accessed and `a[i+1]` in the next step

Temporal because `a[i]` is accessed twice, shortly afterwards

Designing an algorithm for cache efficiency, for instance cache-efficient sorting, will be roughly 3 to 4 times faster than an unoptimized algorithm.

Why do we care about disks?

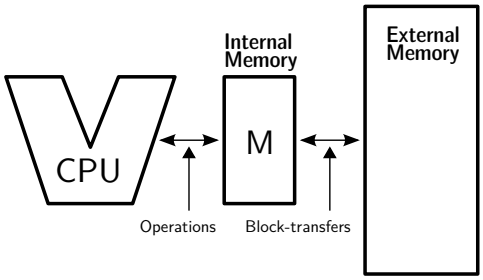
Today's data won't fit into RAM, not even on disk:

Example: Large Hadron Collider – According to CERN's website, LHC generates $\approx$ 100000 DVD's worth of data per year.

Example: Global warming – Companies create terrain models to perform flood simulation. The amount of data points taken can be the difference between "island will be flooded" and "everything will be fine" but will also determine the space requirements: E. g. with data at 30m resolution, modelling Denmark will require $\approx$ 1GB of data, while at 1m resolution, more than 1TB of data will be necessary.

1.2 Models for Studying Memory Hierarchies

1.2.1 The External Memory Model



Data is transferred between external and internal memories in blocks of size B.

Our primary complexity metric is *I/O complexity* ($Q(N)$):

How many blocks are transferred between external and internal memory?

Our secondary complexity metric is *work complexity* ($T(N)$):

How many operations are performed once the data is in internal memory?

I/O-complexity counts the transferred blocks.

1.2.2 The Cache-Oblivious Model

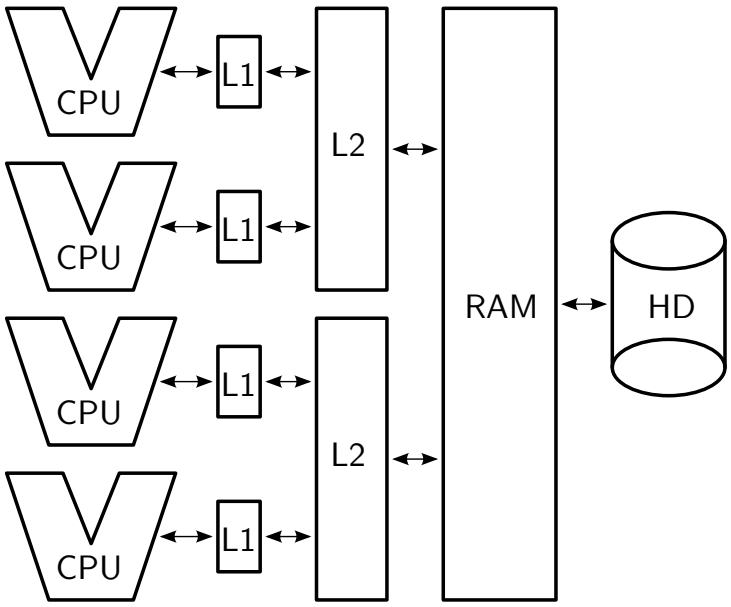
The cache-oblivious model is basically the same as the external memory model, but with an important distinction: In the cache-oblivious model no knowledge of the sizes of M and B is assumed during algorithm design. These parameters are used only during analysis of algorithms.

This makes algorithms harder to design, but will allow them to be efficient regardless of the sizes of M or B and applied to all levels of memory hierarchies.

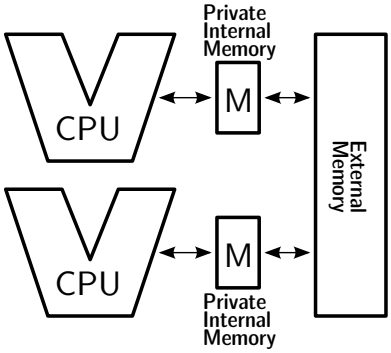
We will cover the cache-oblivious model in December.

1.2.3 The Parallel External Memory (PEM) Model

Parallel architectures increase the complexity:



The parallel external memory (PEM) model uses a simplified memory hierarchy, in which each processor has its own private cache:



We will talk about the PEM model in January.

1.2.4 The Parallel Cache-Oblivious Model

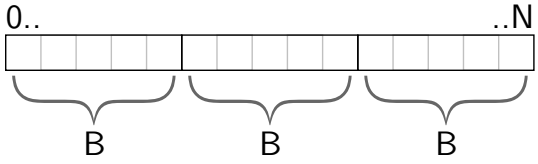
The parallel cache-oblivious model is a parallel model for more complex memory hierarchies. We will cover it at the end of this course.

1.3 I/O-efficient Algorithms and Data Structures

1.3.1 Sequential Scan

```
for (i = 0, i < N, i++) {
     sum += a[i];
}
```

In this example, I/O-complexity can be computed by dividing the input size N by the block size B of a transfer; it is the number of necessary transfers.



I/O-complexity analysis:

The input consists of $\left\lceil \frac{N}{B} \right\rceil$ blocks, and during scan each block is loaded into the internal memory only once.

Therefore, I/O-complexity is $O\left(\frac{N}{B}\right)$ I/Os.

Note that we didn't use any knowledge of M and B in the algorithm, but only during analysis. Thus, this scanning algorithm is also cache-oblivious.

1.4 Sequential Memory Models

1.4.1 Stacks

Stack should support two operations:

- `push(x)`
- `pop()`

A simple implementation of the stack is via a dynamic array, where `push(x)` appends `x` at the end of the array and `pop()`, removes and returns the last element at the array.

Claim 1: N `push(x)` operations take $O\left(\frac{N}{B}\right)$ I/Os.

Proof: Maintain the last block of array elements in internal memory. If the last block is not full, the `push(x)` operation appends `x` at the end of the block resulting in no additional block transfers. When the block is full we can write it out to external memory, clearing the block stored in internal memory, resulting in 1 I/O. Since we must have performed B `push(x)` operations (without any I/Os) to make the block full B `push(x)` operations cost us only 1 I/O. Thus, N `push(x)` operations will result in $O\left(\frac{N}{B}\right)$ I/Os.

Claim 2: N `pop()` operations on a stack of at least N elements takes $O\left(\frac{N}{B}\right)$ I/Os.

Proof: Maintain the last non-empty block of the array in internal memory. The $O\left(\frac{N}{B}\right)$ I/O complexity follows from an argument similar to proof of *Claim 1*.

Note, that if we have a combination of N `push(x)` and `pop()` operations, the above simple strategy of keeping just one block in internal memory does not work to achieve $O\left(\frac{N}{B}\right)$ I/O complexity. Consider the case when we have $B + 1$ items on the stack. This means that B items are stored in external memory and 1 item is stored in internal memory. If we now perform two `pop()` operations, the block in internal memory becomes empty and we load the block the next non-empty block from external memory. If we then perform two `push()` operations, the block in internal memory fills up and we have to write it out. Thus, we end up performing a block transfer after every second operations, resulting in $O(N)$ I/Os after N operations.

Claim 3: A combination of N `push(x)` and `pop()` operations takes $O\left(\frac{N}{B}\right)$ I/Os.

Proof: We slightly adapt the paging strategy as follows: We maintain the last k items in internal memory, where $\frac{B}{4} \leq k \leq \frac{7B}{4}$ using at most two blocks of internal memory. If $k = \frac{B}{4}$ and we perform a `pop()` operation, we load the next full block of the array into internal memory, thus setting $k = \frac{5B}{4}$. If $k = \frac{7B}{4}$ and we perform a `push()` operation, we first write out the full block out to external memory, thus setting $k = \frac{3B}{4}$. Thus, each I/O happens after we performed at least $\frac{B}{2}$ operations and the total I/O complexity an arbitrary combination of any N operations results in at most $\frac{2N}{B} = O(\frac{N}{B})$ I/Os.

For stacks, the *Least-Recently-Used* (LRU) policy is almost as good as the theoretical optimum.

1.4.2 Queues

A queue supports two operations:

- `insert(x)` inserts the item `x`
- `deleteFirst()` removes and returns the first element of the queue

Using the same replacement policy, the I/O complexity is the same as for a stack: $O\left(\frac{1}{B}\right)$ I/Os per element amortized.

Claim: A combination of N `insert(x)` and `delete()` operations on the queue takes $O\left(\frac{N}{B}\right)$ I/Os.

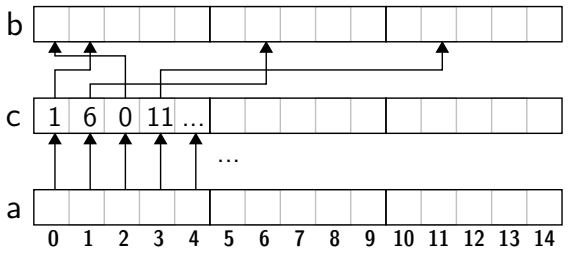
Proof: The proof is similar to the proof of Claim 3. But now we maintain one block of items from the beginning of the array and one block of items from the end of the array in internal memory.

1.4.3 Pointers

So far we have seen traditional algorithms which are I/O-efficient even without any additional modifications.

However, consider the following code snippet, which uses indirect memory access (i. e. pointers).

```
for (i = 0, i < n, i++)
    a[i] = b[c[i]];
```



This code executes $O(N)$ operations, which is the same as scanning in the work-complexity model. However, in the worst case we might read a new block at each iteration, i. e. $O(N) \gg scan(N)$. Therefore, the I/O complexity of this code is $O(N)$, which is much worse than $scan(N) = O\left(\frac{N}{B}\right)$.

operation	I/O-complexity	work-complexity
`scan(N)`	$O\left(\frac{N}{B}\right)$	$O(N)$
`sort(N)`	$O\left(\frac{N}{B} \cdot log_{\frac{M}{B}} \frac{N}{B}\right)$	$O(N \cdot log(N)$

In general: $N \gg sort(N) > scan(N)$

We will see how to process pointers efficiently in the lecture on EM graph algorithms.

1.5 Sorting in the I/O model

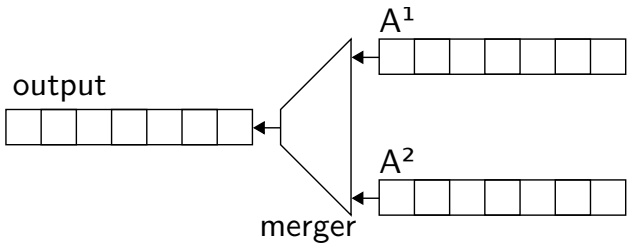
1.5.1 Mergesort

Simple `mergesort(A[1 ... N])`:

```
A1 = mergesort(A[1 ... (N/2)])
A2 = mergesort(A[(N/2+1) ... N])
return merge(A1, A2)
```

Work-complexity is defined by the following recurrence:

$T(N) = 2T\left(\frac{N}{2}\right) + O(N) = O(N \cdot logN)$



Merge:

- Load the first blocks of A^1, A^2
- Choose the smaller one
- If one is empty, copy the other

I/O-complexity of merge:

$scan(|A^1|) + scan(|A^2|) + scan(|output|) = scan\left(\frac{N}{2}\right) + scan\left(\frac{N}{2}\right) + scan\,(N) = O\left(\frac{N}{2B}\right) + O\left(\frac{N}{2B}\right) + O\left(\frac{N}{B}\right) = O\left(\frac{N}{B}\right)$

Then the I/O-complexity of the mergesort is defined by the following recurrence:

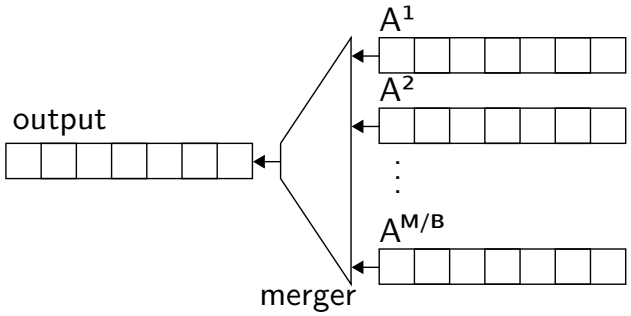
$$Q(N) \begin{cases} 2Q\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\right) & \text{if N} > \text{B} \\ O(1) & \text{if N} < \text{B} \end{cases}$$

$= O\left(\frac{N}{B} \cdot log_2 \frac{N}{B}\right)$

$Q(N) = scan(N)$

The above merging utilizes only 3 blocks of internal memory: 2 input and 1 output.

We can do better with a multiway-mergesort, in which we merge $O(\frac{M}{B})$ sorted arrays at a time:



Multiway-`mergesort(A[1 ... N])`:

```
A1      = mergesort(A[1 ... N/(M/B)])
A2      = mergesort(A[(N/(M/B))+1 ... 2N/(M/B)])
...
A(M/B) = mergesort(A[(N/(M/B))*((M/B)-1)+1 ... N])
return merge(A1, A2, ..., A(M/B))
```

I/O-complexity of multiway-merge(N):

$$\begin{aligned} & scan(|A^1|) + scan(|A^2|) + ... + scan(|A^{\frac{M}{B}}|) + scan(|A|) \\ = \; & O\left(\frac{N}{\frac{M}{B}}\right) + O\left(\frac{N}{\frac{M}{B}}\right) + ... + O\left(\frac{N}{\frac{M}{B}}\right) + O\left(\frac{N}{B}\right) \\ = \; & O\left(\frac{N}{B}\right) \text{ IOs} \end{aligned}$$

$O(\frac{N}{B} \cdot log_{\frac{M}{B}} \frac{N}{B})$

Example with

- input size $N = 4\text{GB} = 2^{32}\text{Byte}$
- cache line size $B = 64\text{Byte} = 2^{6}\text{Byte}$
- cache size $M = 8\text{MB} = 2^{23}\text{Byte}$
- memory throughput $\approx 4\text{GB/s}$ ($scan(N)$ takes 1s)

Thus $\frac{M}{B} = \frac{2^{23}}{2^6} = 2^{17}$and $\frac{N}{B} = \frac{2^{32}}{2^6} = 2^{26}$.

Using the given sizes, 2-way-mergesort will incur $O\left(\frac{N}{B} log_2 \frac{N}{B}\right) = (2^{26}) log_2(2^{26}) = 26 \cdot (2^{26})$ I/Os.

Rough estimate of data transfer:

2-way-mergesort:

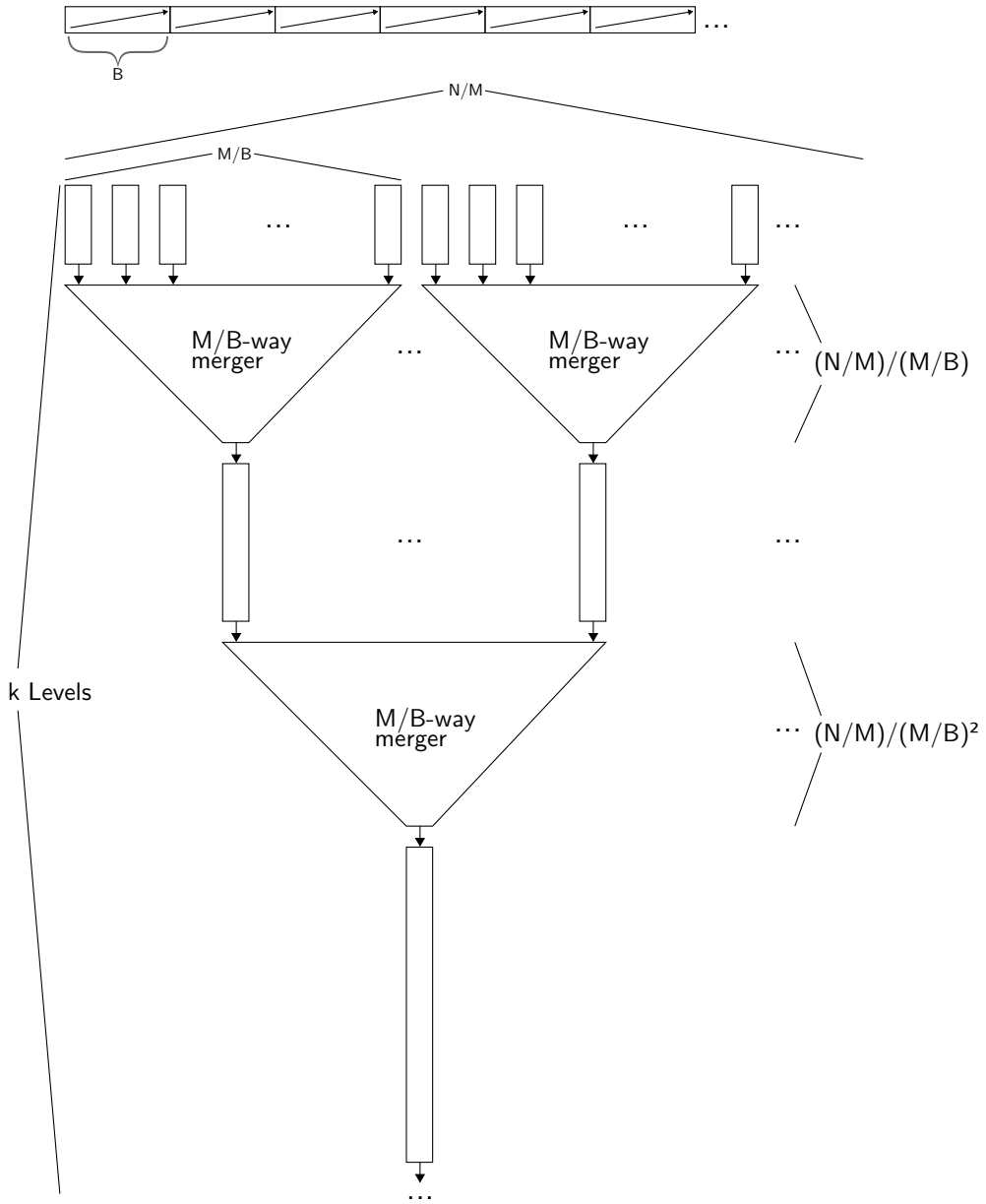
$$\begin{aligned}\left(\frac{N}{B}\right) \cdot log_2\left(\frac{N}{B}\right) &= scan(N) \cdot log_2\left(\frac{N}{B}\right) \\ &= 1s \cdot log_2\left(\frac{N}{B}\right) \\ &= 1s \cdot log_2\left(2^{26}\right) \\ &= 26s\end{aligned}$$

$\frac{M}{B}$-way-multisort:

$$\begin{aligned}\left(\frac{N}{B}\right) \cdot log_{\frac{M}{B}}\left(\frac{N}{B}\right) &= scan(N) \cdot log_{\frac{M}{B}}\left(\frac{N}{B}\right) \\ &= 1s \cdot log_{2^{17}}\left(2^{26}\right) \\ &= 1s \cdot \frac{log 2^{26}}{log 2^{17}} \\ &\leq 2s\end{aligned}$$

How to compute the complexity?

Use a simplified model, in which the input of size N is splitted into chunks of size B.



$N = (\frac{M}{B})^k \cdot M$;

After k levels, we'll have

$$\begin{aligned} \frac{\frac{N}{M}}{\left(\frac{M}{B}\right)^k} &= 1 \\ \frac{N}{M} &= \left(\frac{M}{B}\right)^k \\ k &= log_{\frac{M}{B}} \frac{N}{M} \end{aligned}$$

The I/O-complexity of multiway-mergesort is defined by the following recurrence:

$$Q(N) \begin{cases} \frac{M}{B} Q(\frac{N}{\frac{M}{B}}) + O(\frac{N}{B}) & \text{if N > B} \\ O(1) & \text{if N < B} \end{cases}$$

$= O(\frac{N}{B} \cdot log_{\frac{M}{B}} \frac{N}{B})$

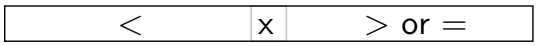
I/O complexity:

$$\begin{aligned} & k \cdot \text{I/O complexity of all merging} + \text{sorting each M-sized chunk} \\ = \; & merge\left(\frac{M^2}{B}\right) \cdot \frac{\frac{N}{M}}{\frac{M}{B}} + scan(N) \\ = \; & k \cdot O\left(\frac{N}{B}\right) + O\left(\frac{N}{B}\right) \\ = \; & \left(log_{\frac{M}{B}} \frac{N}{M}\right) \cdot O\left(\frac{N}{B}\right) + O\left(\frac{N}{B}\right) \\ = \; & O\left(\frac{N}{B} \cdot \left(log_{\frac{M}{B}} \frac{N}{M} + 1\right)\right) \\ = \; & O\left(\frac{N}{B} \cdot \left(log_{\frac{M}{B}} \frac{N}{M} + log_{\frac{M}{B}} \frac{M}{B}\right)\right) \\ = \; & O\left(\frac{N}{B} \cdot \left(log_{\frac{M}{B}} \left(\frac{N}{M} \cdot \frac{M}{B}\right)\right)\right) \\ = \; & O\left(\frac{N}{B} \cdot \left(log_{\frac{M}{B}} \frac{N}{B}\right)\right) \end{aligned}$$

1.5.2 Quicksort

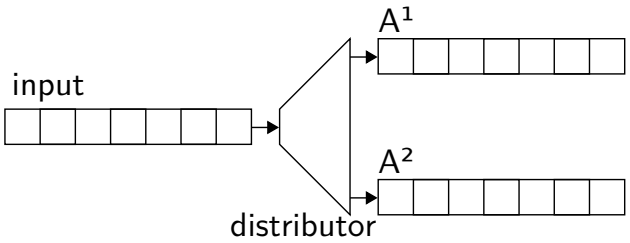
2-way-quickesort:

`quicksort(A[1 ... N])`:




```
x <- pivot(A) // Choose x
A1[1...l] = A[i] such that A[i] < x
A1[l...r] = A[i] such that A[i] >= x
return qsort(A1) : qsort(A2)
```

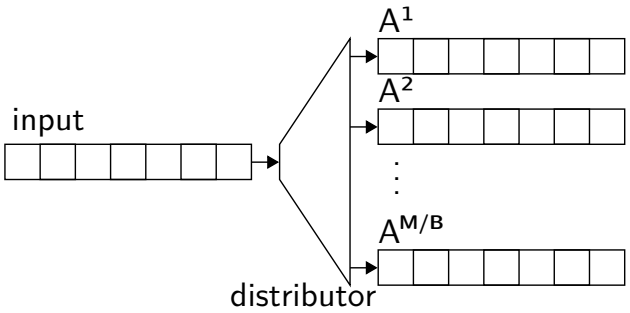
Work-complexity: $T(N) = 2T(\frac{N}{2}) + O(N) = O(N \cdot logN)$



We only use 3 blocks of internal memory: 1 for input and 2 for output.

We can do better by distributing $\frac{M}{B}$-ways!

Multiway-quicksort:



Recurrence:

$Q(N) = \frac{M}{B} \cdot Q(\frac{N}{\frac{M}{B}}) + O(\frac{N}{B}) = O(\frac{N}{B} \cdot log_{\frac{M}{B}} \frac{N}{B})$

We will discuss $\frac{M}{B}$-way distribution sort in the next lecture.