External Sorting

- Sort n records/elements that reside on a disk.
- Space needed by the n records is very large.
  - n is very large, and each record may be large or small.
  - n is small, but each record is very large.
- So, not feasible to input the n records, sort, and output in sorted order.

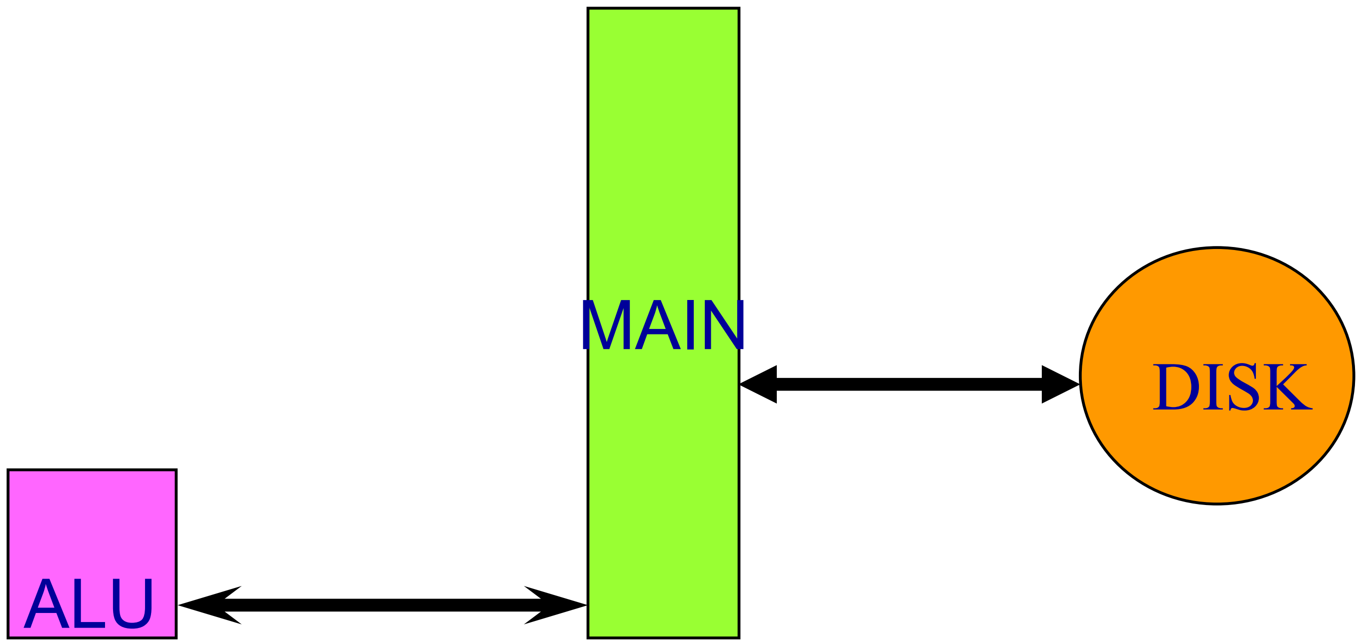
Small n But Large File

- Input the record keys.
- Sort the n keys to determine the sorted order for the n records.
- Permute the records into the desired order (possibly several fields at a time).
- We focus on the case: large n, large file.

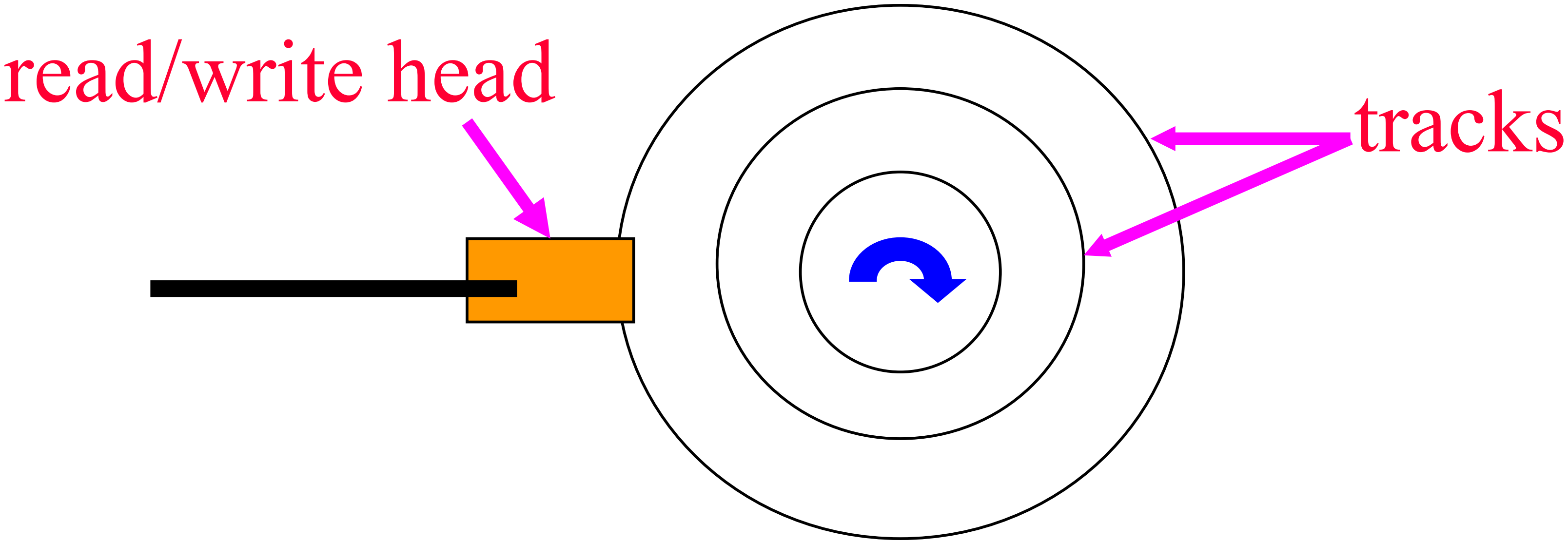
New Data Structures/Concepts

- Tournament trees.
- Huffman trees.
- Double-ended priority queues.
- Buffering.
- Ideas also may be used to speed algorithms for small instances by using cache more efficiently.

External Sort Computer Model

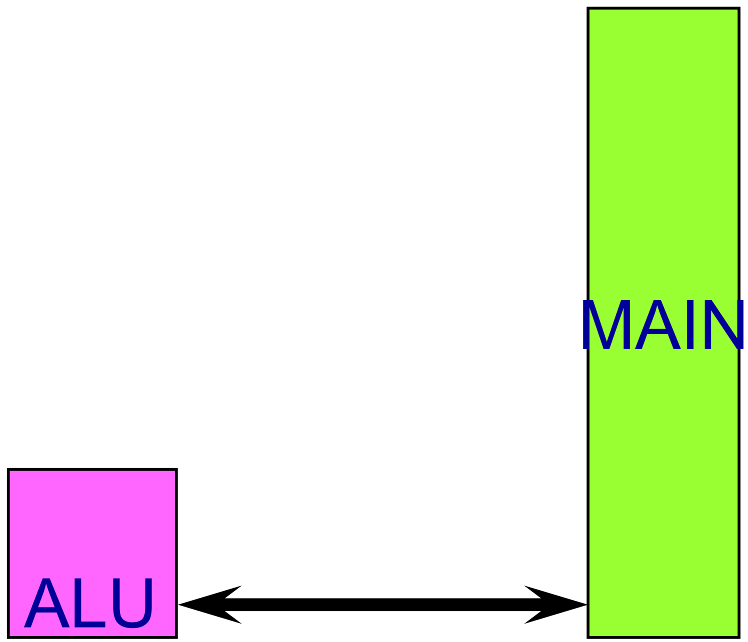


Disk Characteristics



- Seek time
  - Approx. 100,000 arithmetics
- Latency time
  - Approx. 25,000 arithmetics
- Transfer time
- Data access by block

Traditional Internal Memory Model

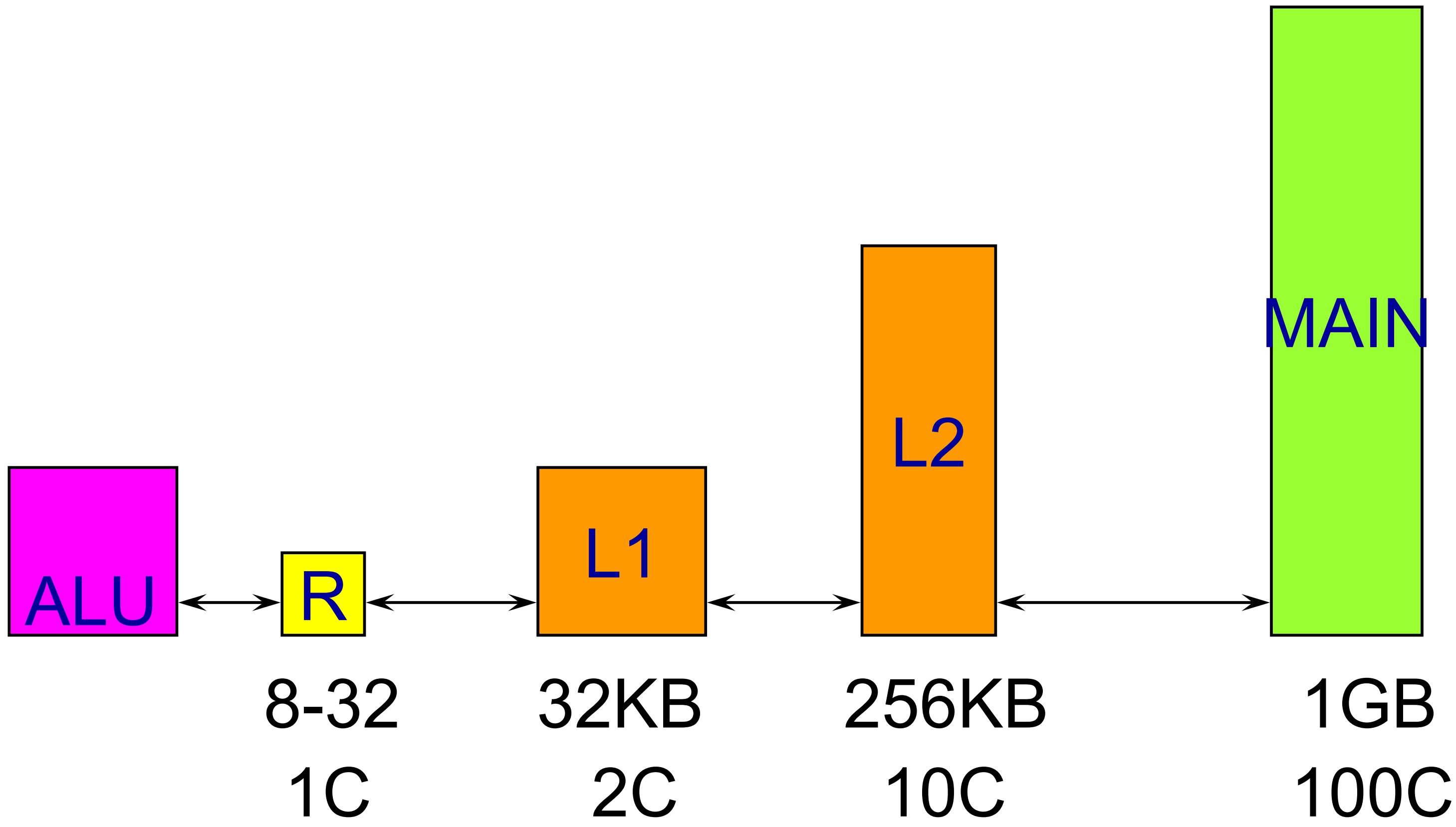


Matrix Multiplication

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      c[i][j] += a[i][k] * b[k][j];
```

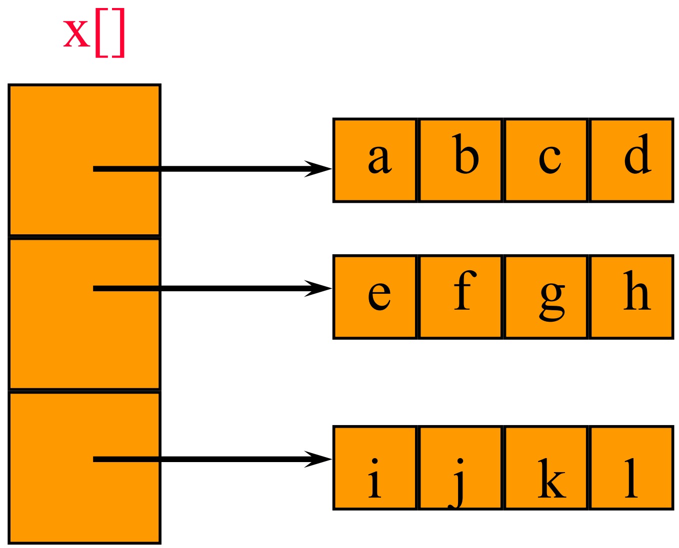
- ijk, ikj, jik, jki, kij, kji orders of loops yield same result.
- All perform same number of operations.
- But run time may differ significantly!

More Accurate Memory Model



2D Array Representation In Java, C, and C++

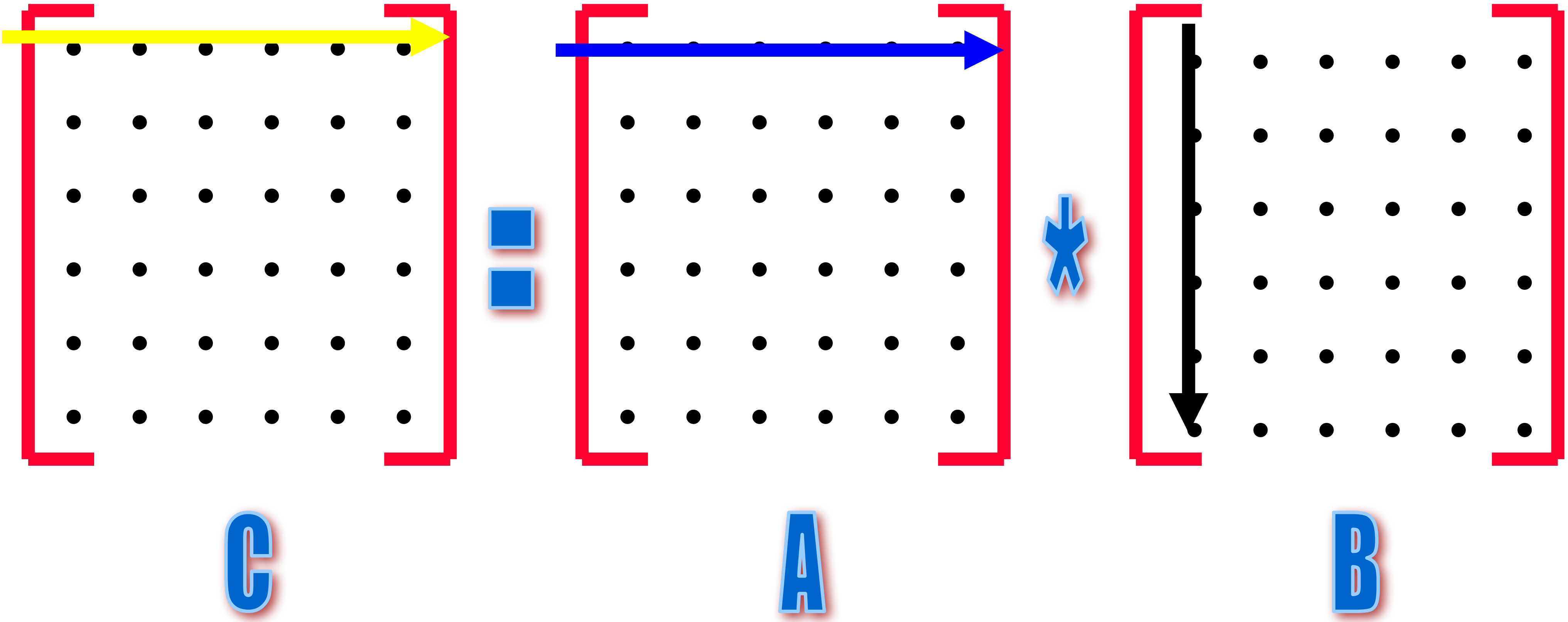
int x[3][4];



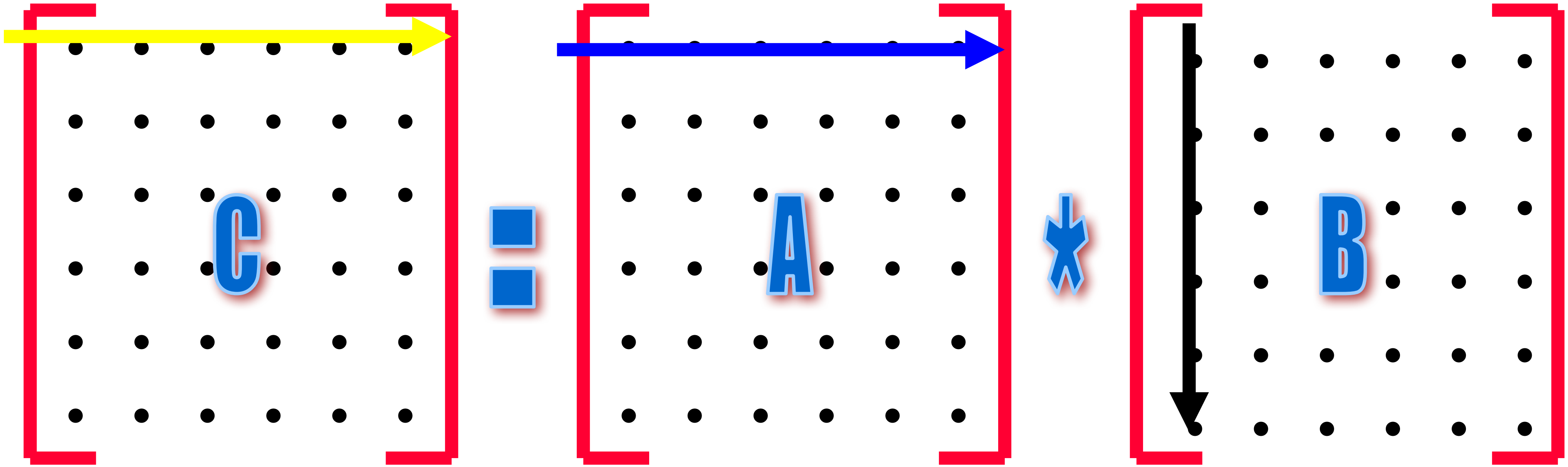
Array of Arrays Representation

ijk Order

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
```



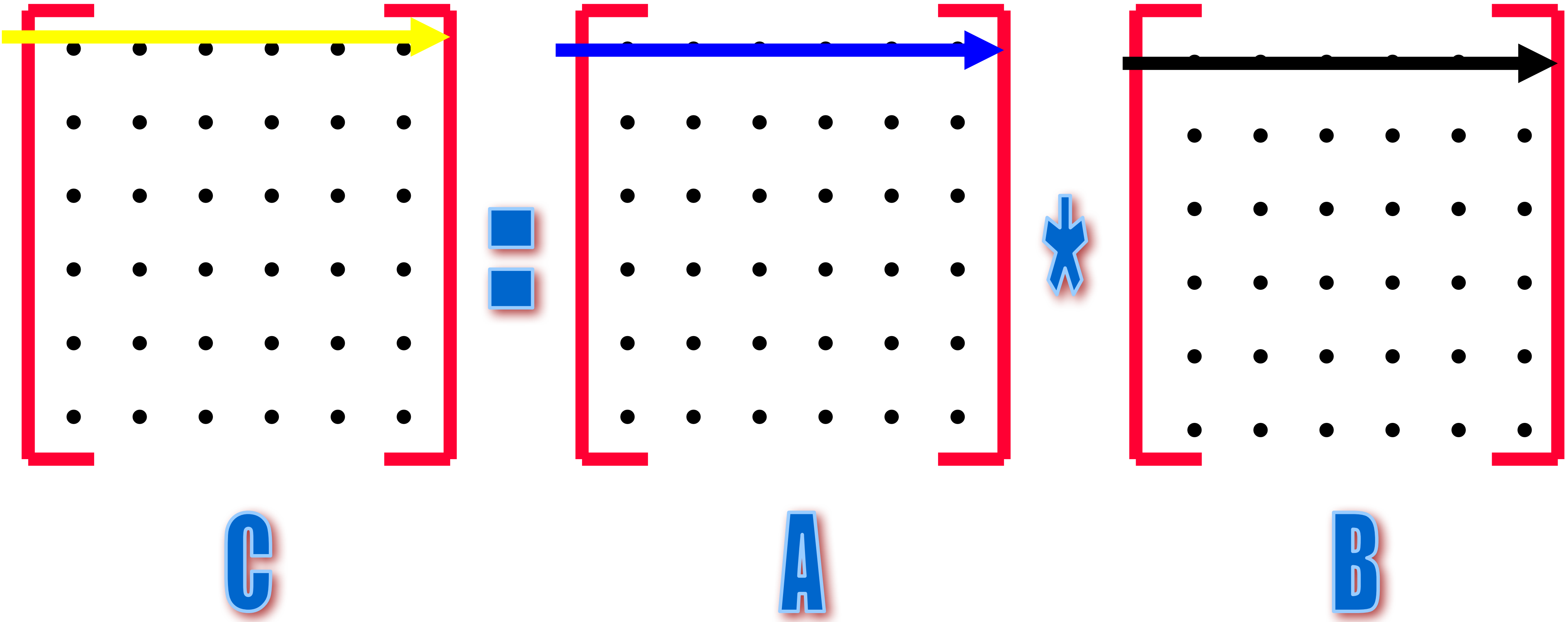
ijk Analysis



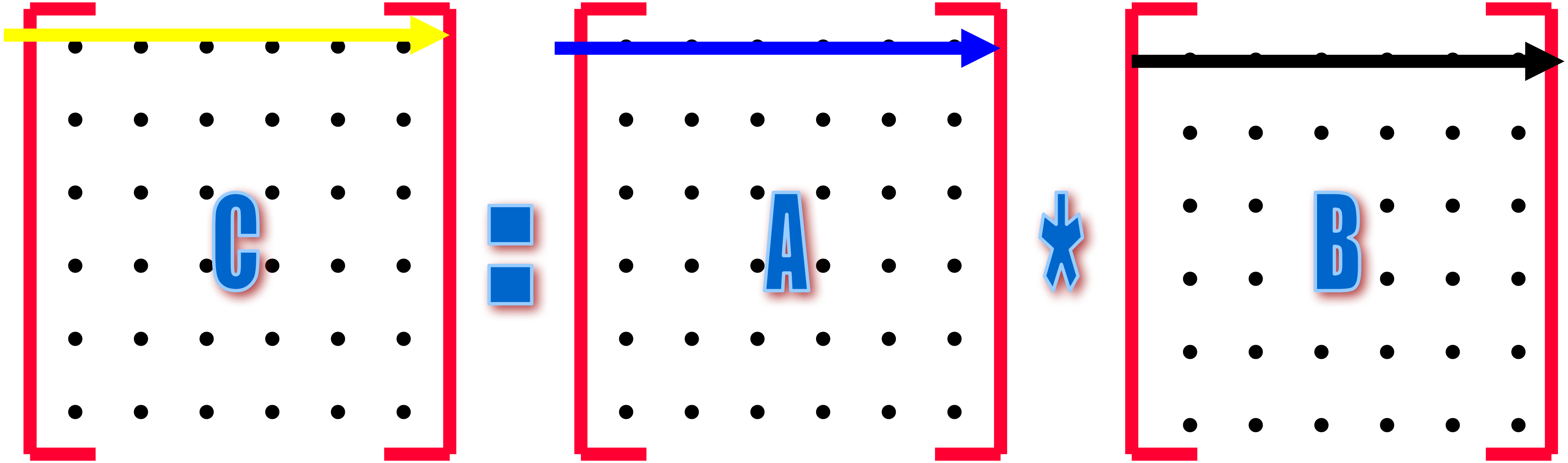
- Block size = width of cache line = w.
- Assume one-level cache.
- C => n^2/w cache misses.
- A => n^3/w cache misses, when n is large.
- B => n^3 cache misses, when n is large.
- Total cache misses = $n^3/w(1/n + 1 + w)$.

ikj Order

```
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++)
    for (int j = 0; j < n; j++)
      c[i][j] += a[i][k] * b[k][j];
```



ikj Analysis



- C => n^3/w cache misses, when n is large.
- A => n^2/w cache misses.
- B => n^3/w cache misses, when n is large.
- Total cache misses = $n^3/w(2 + 1/n)$.

ijk Vs. ikj Comparison

- ijk cache misses $= n^3/w(1/n + 1 + w)$.
- ikj cache misses $= n^3/w(2 + 1/n)$.
- ijk/ikj $\sim (1 + w)/2$, when n is large.
- $w = 4$ (32-byte cache line, double precision data)
  - ratio ~ 2.5.
- $w = 8$ (64-byte cache line, double precision data)
  - ratio ~ 4.5.
- $w = 16$ (64-byte cache line, integer data)
  - ratio ~ 8.5.

Prefetch

- Prefetch can hide memory latency
- Successful prefetch requires ability to predict a memory access much in advance
- Prefetch cannot reduce energy as prefetch does not reduce number of memory accesses

External Sort Methods

- Base the external sort method on a fast internal sort method.
- Average run time
  - Quick sort
- Worst-case run time
  - Merge sort

Internal Quick Sort

- To sort a large instance, select a pivot element from out of the n elements.
- Partition the n elements into 3 groups left, middle and right.
- The middle group contains only the pivot element.
- All elements in the left group are <= pivot.
- All elements in the right group are >= pivot.
- Sort left and right groups recursively.
- Answer is sorted left group, followed by middle group followed by sorted right group.

Internal Quick Sort

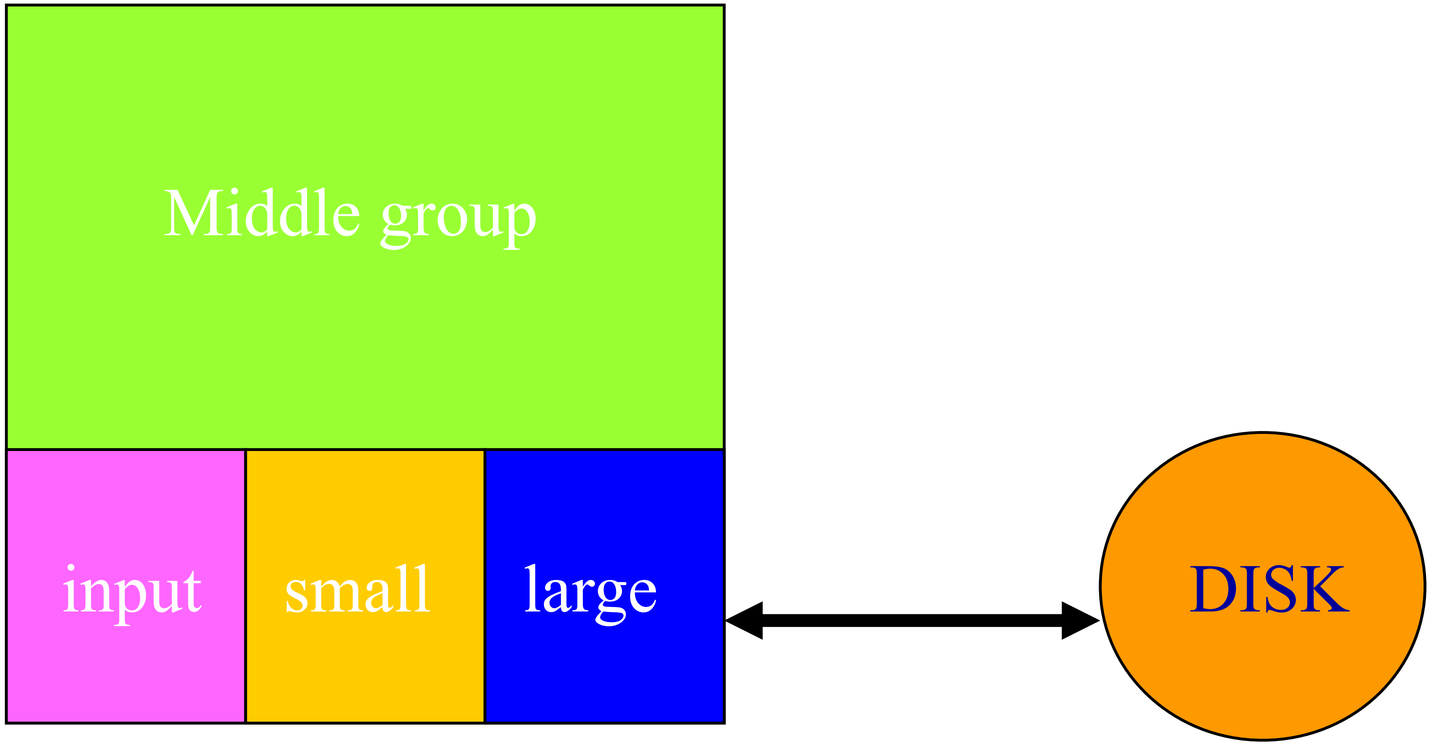
6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot.

2	5	4	1	3	6	7	9	10	11	8

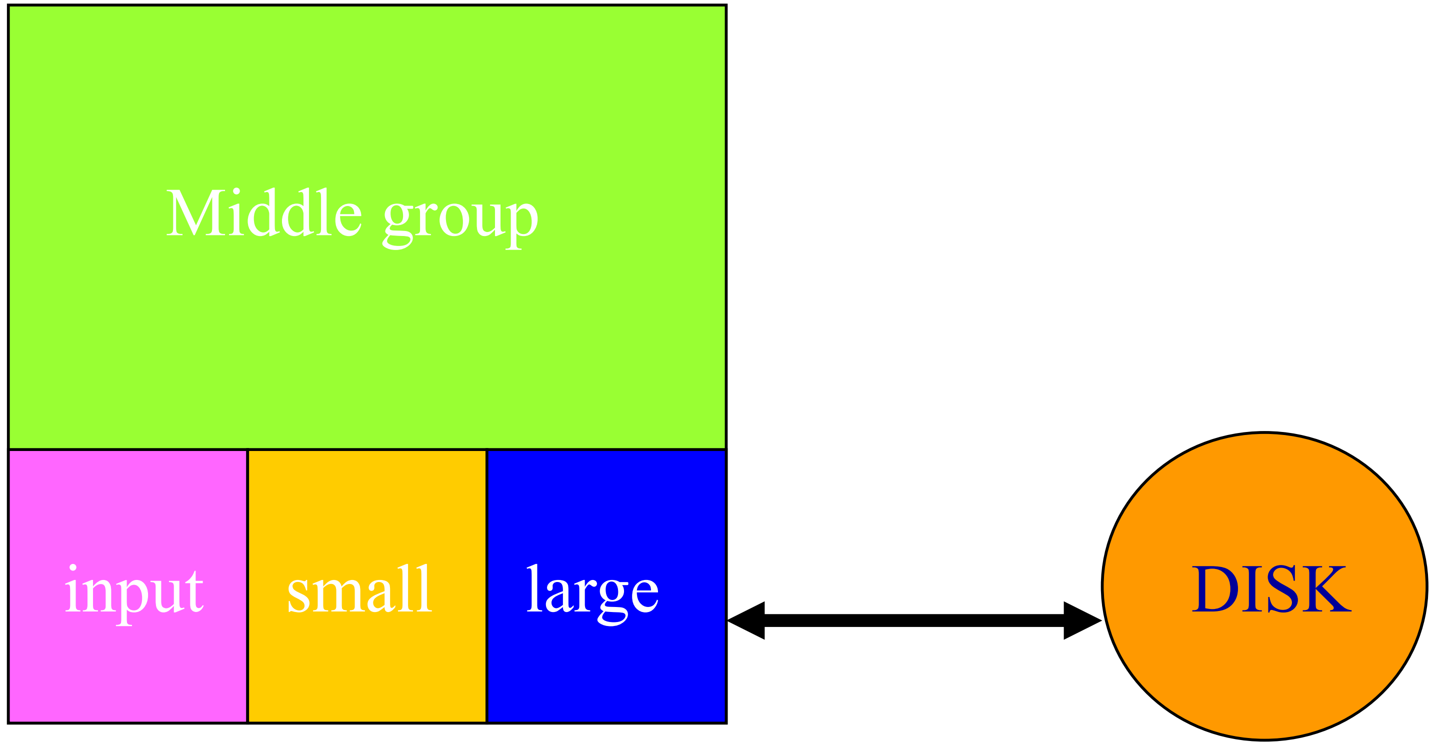
Sort left and right groups recursively.

Quick Sort – External Adaptation



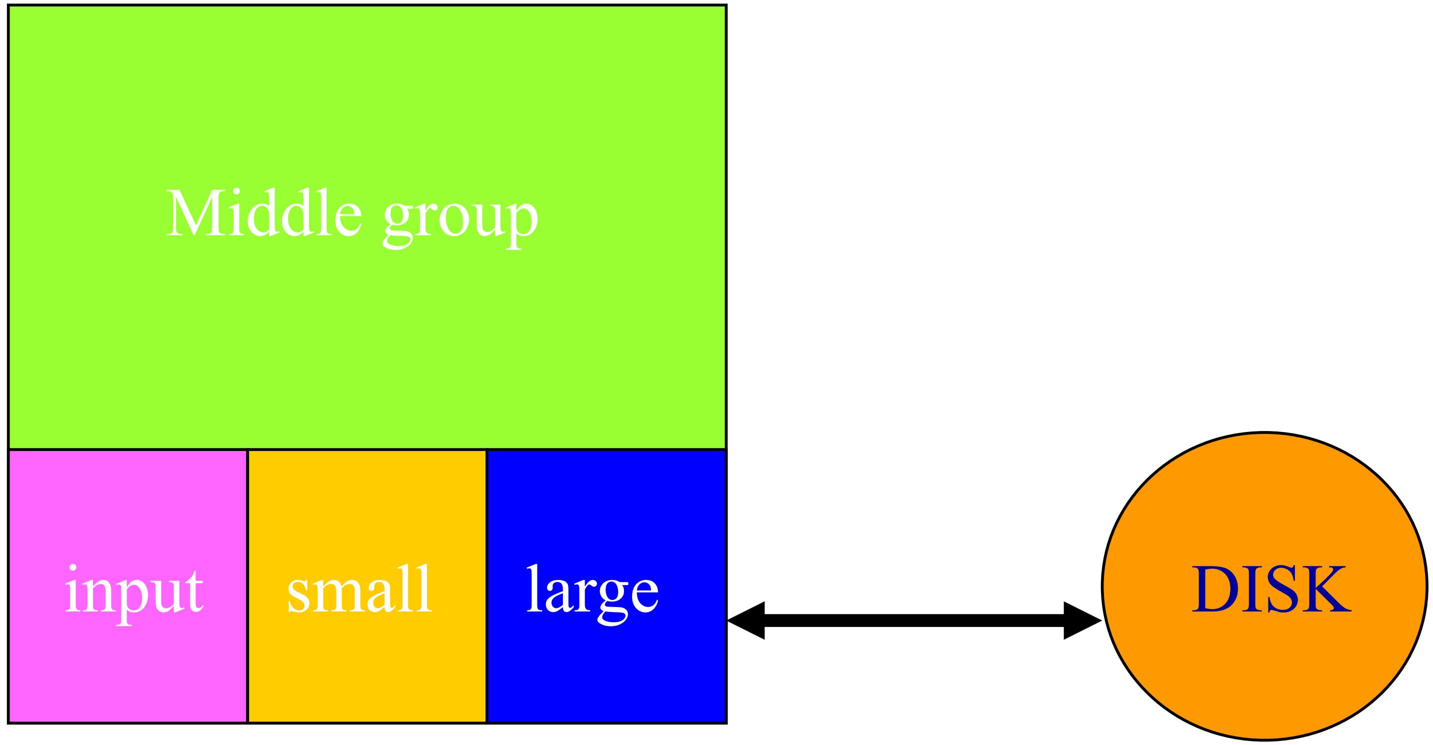
- 3 input/output buffers
  - input, small, large
- rest is used for middle group

Quick Sort – External Adaptation



- fill middle group from disk
- if next record <= middle_{min} send to small
- if next record >= middle_{max} send to large
- else remove middle_{min} or middle_{max} from middle and add new record to middle group

Quick Sort – External Adaptation



- Fill input buffer when it gets empty.
- Write small/large buffer when full.
- Write middle group in sorted order when done.
- Double-ended priority queue.

External Sorting

- Adapt fastest internal-sort methods.
- ✓ Quick sort …best average run time.
- Merge sort … best worst-case run time.

Internal Merge Sort Review

- Phase 1
  - Create initial sorted segments
    - Natural segments
    - Insertion sort
- Phase 2
  - Merge pairs of sorted segments, in merge passes, until only 1 segment remains.

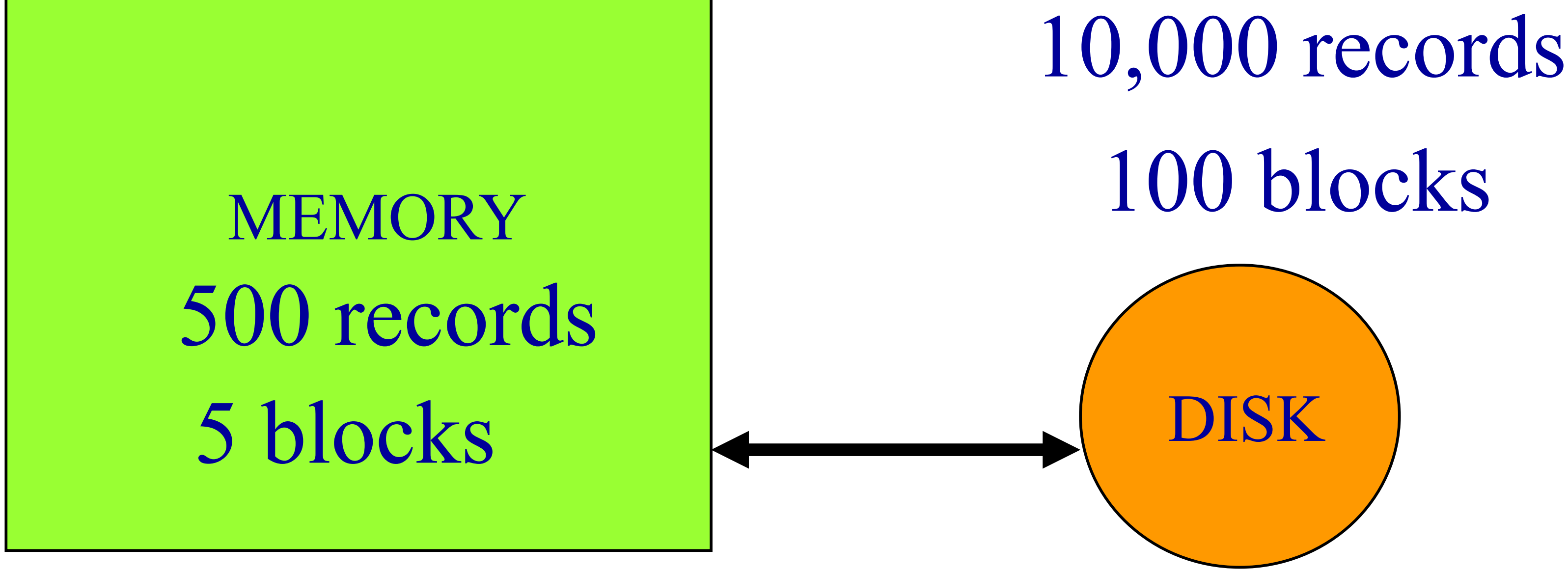
External Merge Sort

- Sort 10,000 records.
- Enough memory for 500 records.
- Block size is 100 records.
- t_{IO} = time to input/output 1 block
  (includes seek, latency, and transmission times)
- t_{IS} = time to internally sort 1 memory load
- t_{IM} = time to internally merge 1 block load

External Merge Sort

- Two phases.
  - Run generation.
    - A run is a sorted sequence of records.
  - Run merging.

Run Generation



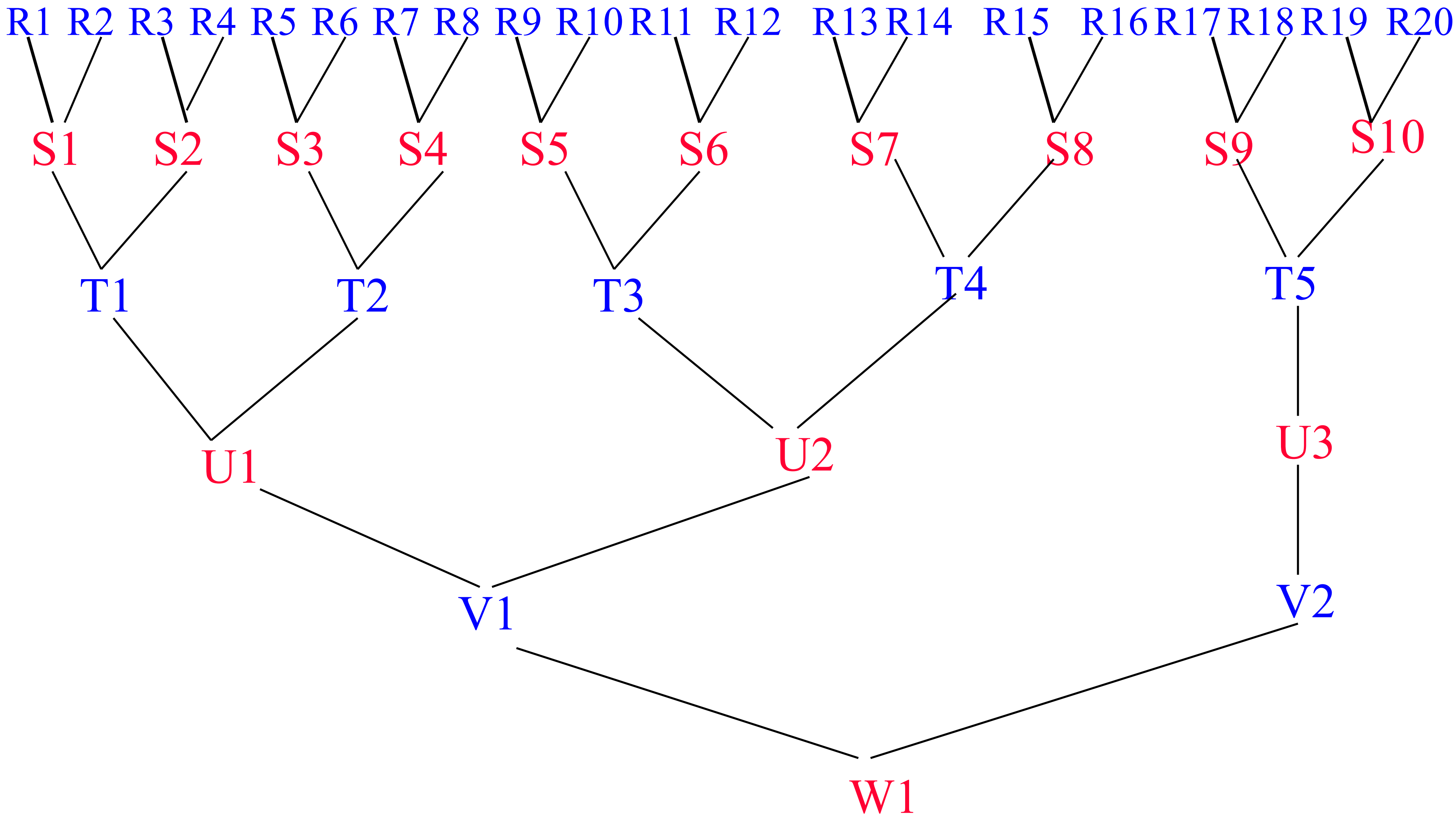
- Input 5 blocks.
- Sort.
- Output as a run.
- Do 20 times.

- $5t_{IO}$
- t_{IS}
- $5t_{IO}$
- $200t_{IO} + 20t_{IS}$

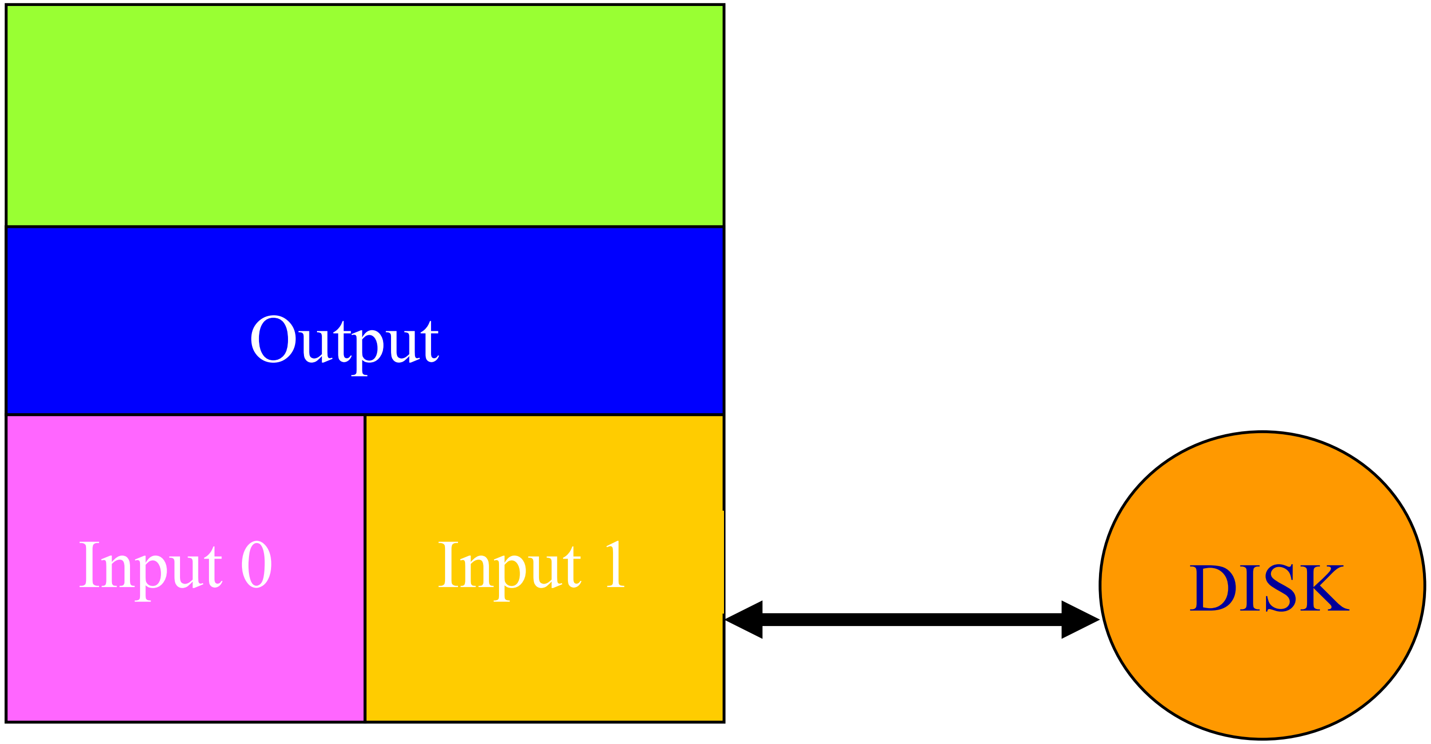
Run Merging

- Merge Pass.
  - Pairwise merge the 20 runs into 10.
  - In a merge pass all runs (except possibly one) are pairwise merged.
- Perform 4 more merge passes, reducing the number of runs to 1.

Merge 20 Runs



Merge R1 and R2



- Fill I0 (Input 0) from R1 and I1 from R2.
- Merge from I0 and I1 to output buffer.
- Write whenever output buffer full.
- Read whenever input buffer empty.

Time To Merge R1 and R2

- Each is 5 blocks long.
- Input time $= 10t_{IO}$.
- Write/output time $= 10t_{IO}$.
- Merge time $= 10t_{IM}$.
- Total time $= 20t_{IO} + 10t_{IM}$.

Time For Pass 1 (R $\rightarrow$ S)

- Time to merge one pair of runs
  $= 20t_{IO} + 10t_{IM}$.
- Time to merge all 10 pairs of runs
  $= 200t_{IO} + 100t_{IM}$.

Time To Merge S1 and S2

- Each is 10 blocks long.
- Input time $= 20t_{IO}$.
- Write/output time $= 20t_{IO}$.
- Merge time $= 20t_{IM}$.
- Total time $= 40t_{IO} + 20t_{IM}$.

Time For Pass 2 (S $\rightarrow$ T)

- Time to merge one pair of runs
  $= 40t_{IO} + 20t_{IM}$.
- Time to merge all 5 pairs of runs
  $= 200t_{IO} + 100t_{IM}$.

Time For One Merge Pass

- Time to input all blocks $= 100t_{IO}$.
- Time to output all blocks $= 100t_{IO}$.
- Time to merge all blocks $= 100t_{IM}$.
- Total time for a merge pass $= 200t_{IO} + 100t_{IM}$.

Total Run-Merging Time

- (time for one merge pass) * (number of passes)

  = (time for one merge pass)

  * ceil($\log_2$(number of initial runs))

  = $(200t_{IO} + 100t_{IM})$ * ceil($\log_2(20)$)

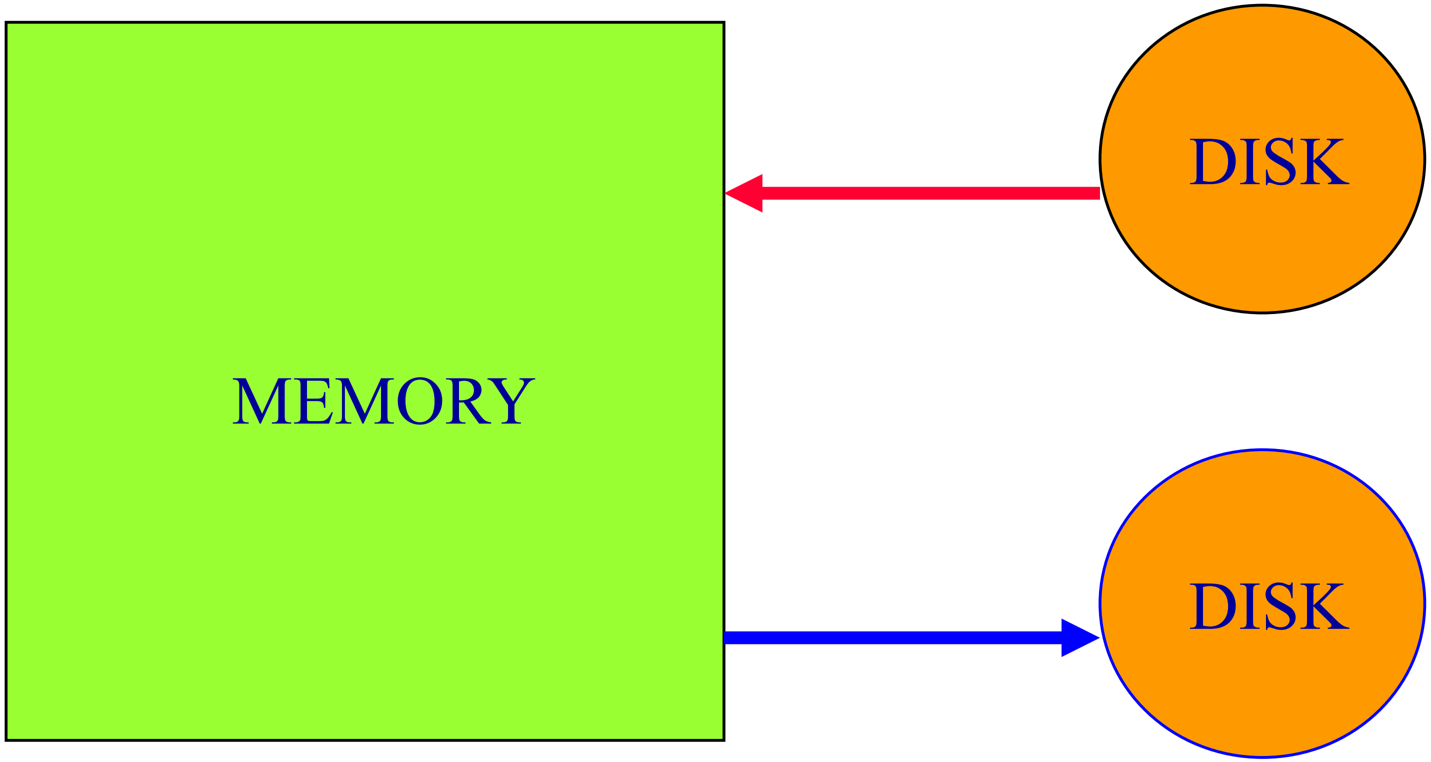
  = $(200t_{IO} + 100t_{IM})$ * 5

Factors In Overall Run Time

- Run generation. $200t_{IO} + 20t_{IS}$
  - Internal sort time.
  - Input and output time.
- Run merging. $(200t_{IO} + 100t_{IM}) * \text{ceil}(\log_2(20))$
  - Internal merge time.
  - Input and output time.
  - Number of initial runs.
  - Merge order (number of merge passes is determined by number of runs and merge order)

Improve Run Generation

- Overlap input, output, and internal sorting.

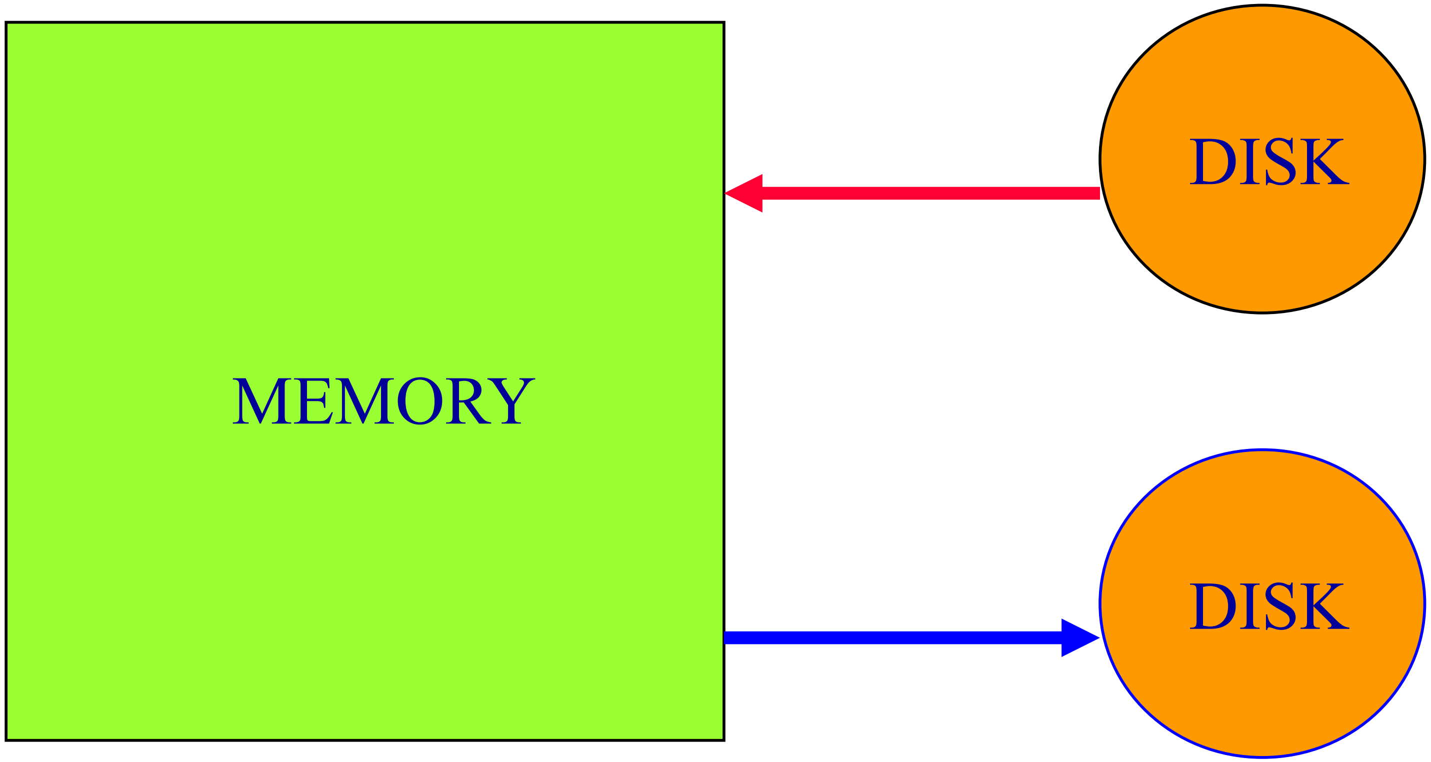


Improve Run Generation

- Generate runs whose length (on average) exceeds memory size.
- Equivalent to reducing number of runs generated.

Improve Run Merging

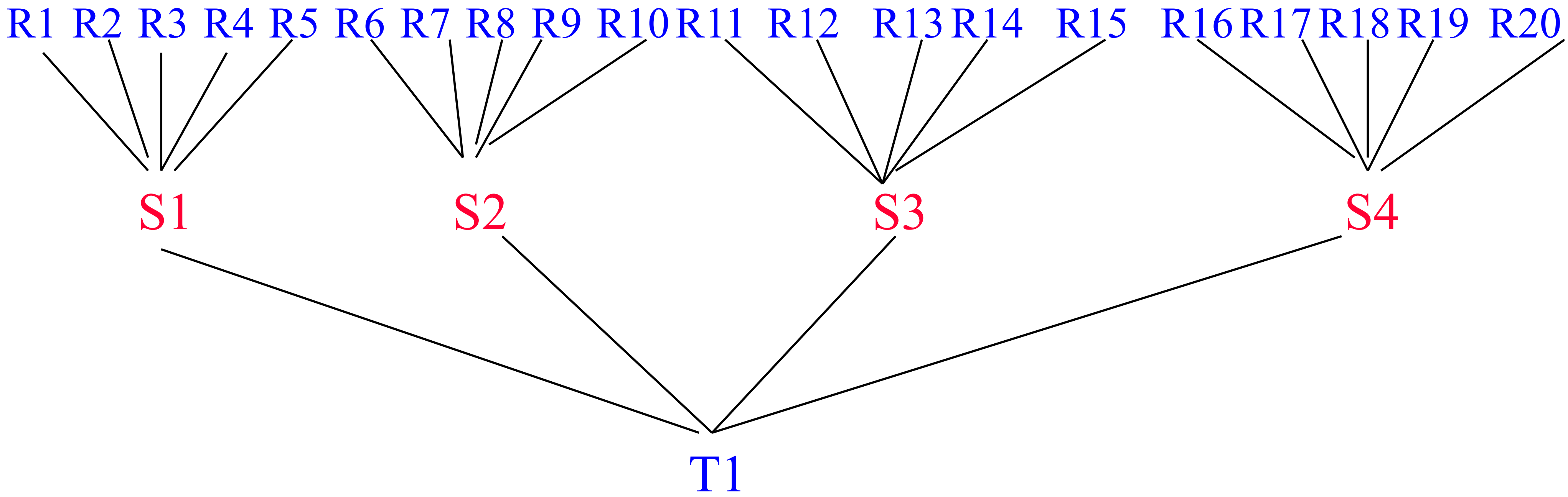
- Overlap input, output, and internal merging.



Improve Run Merging

- Reduce number of merge passes.
  - Use higher-order merge.
  - Number of passes
    $= \text{ceil}(\log_k(\text{number of initial runs}))$
    where k is the merge order.

Merge 20 Runs Using 5-Way Merging

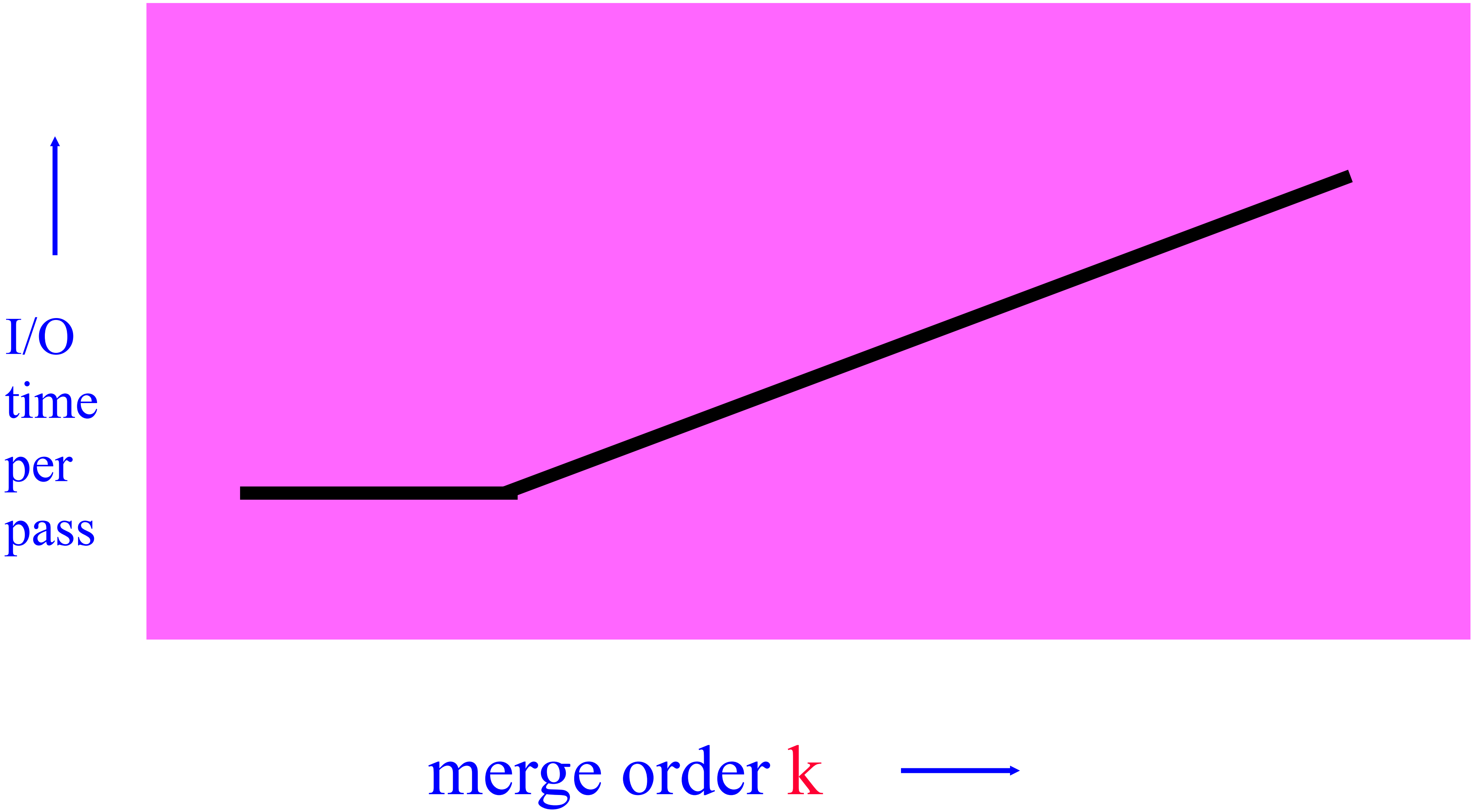


Number of passes = 2

I/O Time Per Merge Pass

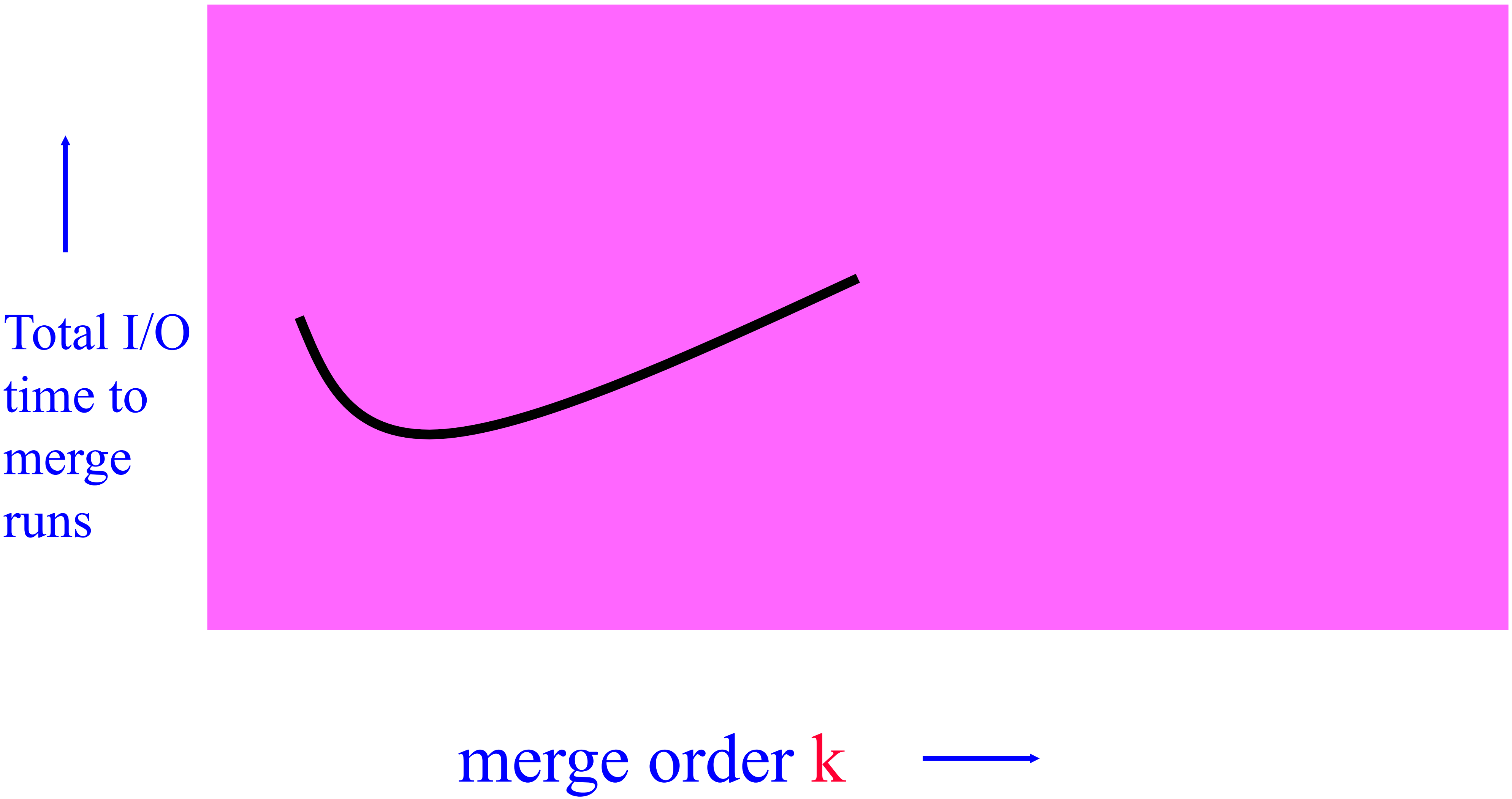
- Number of input buffers needed is linear in merge order k.
- Since memory size is fixed, block size decreases as k increases (after a certain k).
- So, number of blocks increases.
- So, number of seek and latency delays per pass increases.

I/O Time Per Merge Pass

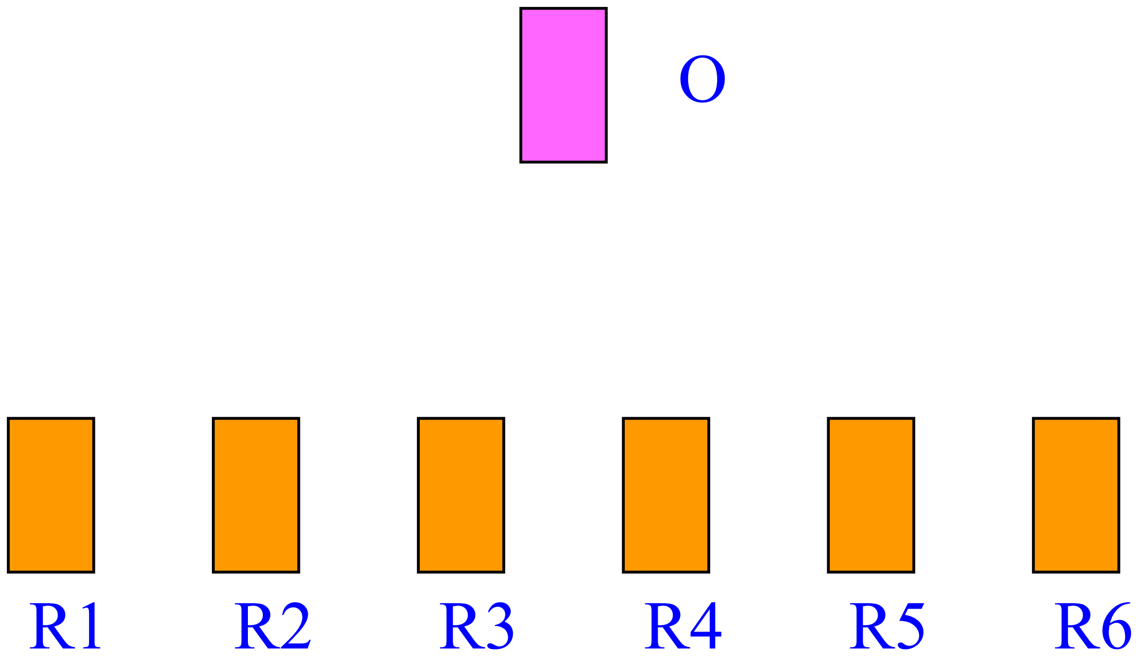


Total I/O Time To Merge Runs

- (I/O time for one merge pass) * ceil($\log_k$(number of initial runs))

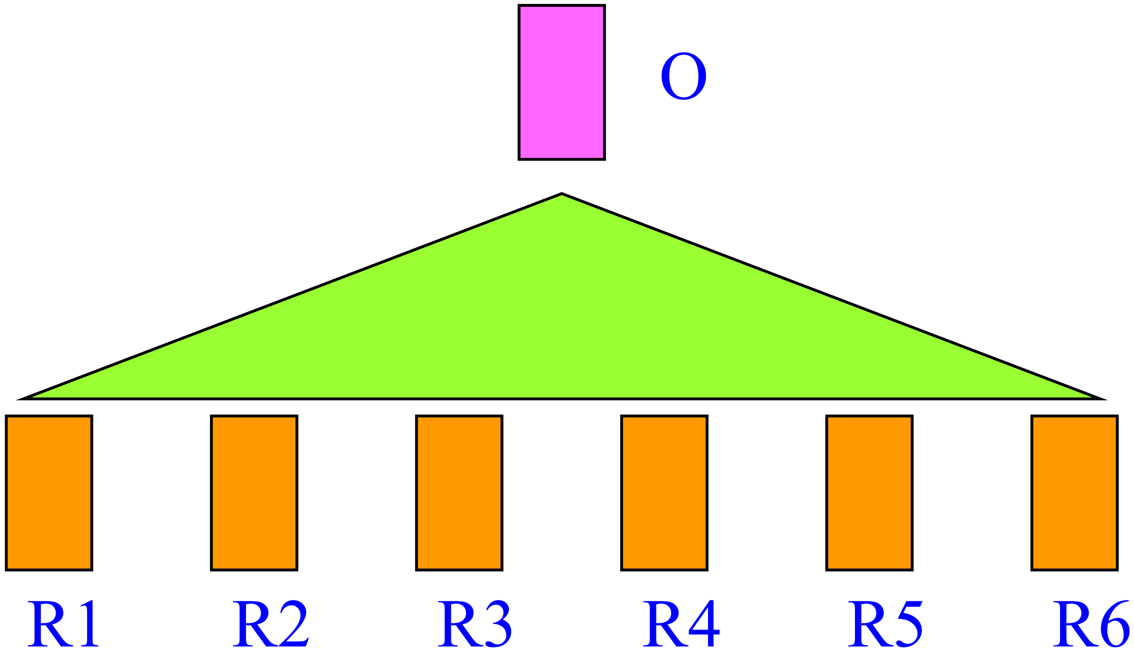


Internal Merge Time



- Naïve way => $k - 1$ compares to determine next record to move to the output buffer.
- Time to merge n records is $c(k - 1)n$, where c is a constant.
- Merge time per pass is $c(k - 1)n$.
- Total merge time is $c(k - 1)n\log_k r \sim cn(k/\log_2 k) \log_2 r$.

Merge Time Using A Tournament Tree



- Time to merge n records is $dn\log_2 k$, where d is a constant.
- Merge time per pass is $dn\log_2 k$.
- Total merge time is $(dn\log_2 k) \log_k r = dn\log_2 r$.