Welcome To . . .

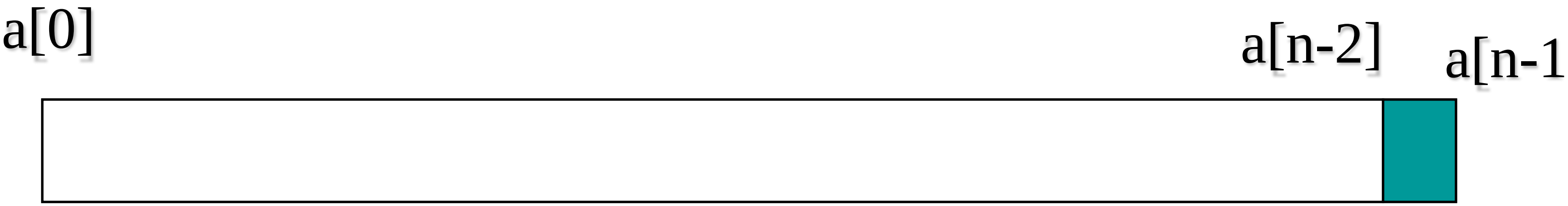
Data Structures

王惠嘉

Sorting

- Rearrange n elements into ascending order.
- 7, 3, 6, 2, 1 ➔ 1, 2, 3, 6, 7

Insertion Sort



- n <= 1 ➔ already sorted. So, assume n > 1.
- a[0:n-2] is sorted recursively.
- a[n-1] is inserted into the sorted a[0:n-2].
- Complexity is $O(n^2)$.
- Usually implemented nonrecursively (see text).

Quick Sort

- When n <= 1, the list is sorted.
- When n > 1, select a pivot element from out of the n elements.
- Partition the n elements into 3 segments left, middle and right.
- The middle segment contains only the pivot element.
- All elements in the left segment are <= pivot.
- All elements in the right segment are >= pivot.
- Sort left and right segments recursively.
- Answer is sorted left segment, followed by middle segment followed by sorted right segment.

Example

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 segments recursively.

Choice Of Pivot

- Pivot is leftmost element in list that is to be sorted.
  - When sorting a[6:20], use a[6] as the pivot.
  - Text implementation does this.
- Randomly select one of the elements to be sorted as the pivot.
  - When sorting a[6:20], generate a random number r in the range [6, 20]. Use a[r] as the pivot.

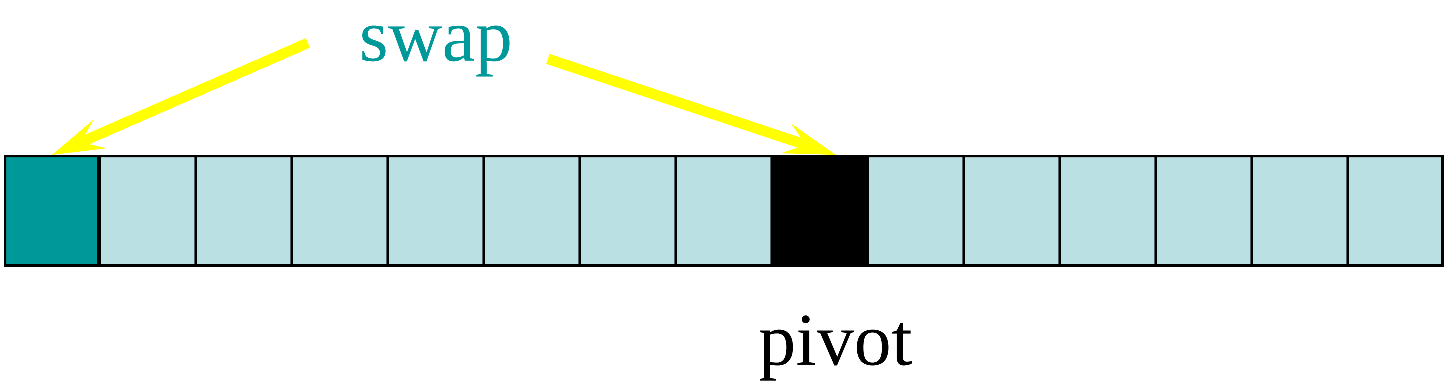
Choice Of Pivot

- Median-of-Three rule. From the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot.
  - When sorting a[6:20], examine a[6], a[13] ((6+20)/2), and a[20]. Select the element with median (i.e., middle) key.
  - If a[6].key = 30, a[13].key = 2, and a[20].key = 10, a[20] becomes the pivot.
  - If a[6].key = 3, a[13].key = 2, and a[20].key = 10, a[6] becomes the pivot.

Choice Of Pivot

- If a[6].key = 30, a[13].key = 25, and a[20].key = 10, a[13] becomes the pivot.

- When the pivot is picked at random or when the median-of-three rule is used, we can use the quick sort code of the text provided we first swap the leftmost element and the chosen pivot.



Partitioning Example Using Additional Array

a

6	2	8	5	11	10	4	1	9	7	3

b

2	5	4	1	3	6	7	9	10	11	8

Sort left and right segments recursively.

In-Place Partitioning Example

a	6	2	8	5	11	10	4	1	9	7	3

a	6	2	3	5	11	10	4	1	9	7	8

a	6	2	3	5	1	10	4	11	9	7	8

a	6	2	3	5	1	4	10	11	9	7	8

bigElement is not to left of smallElement, terminate process. Swap pivot and smallElement.

a	4	2	3	5	1	6	10	11	9	7	8

Merge Sort

- Partition the $n > 1$ elements into two smaller instances.
- First ceil(n/2) elements define one of the smaller instances; remaining floor(n/2) elements define the second smaller instance.
- Each of the two smaller instances is sorted recursively.
- The sorted smaller instances are combined using a process called merge.
- Complexity is $O(n \log n)$.
- Usually implemented nonrecursively.

Merge Two Sorted Lists

- A = (2, 5, 6)
  B = (1, 3, 8, 9, 10)
  C = ()
- Compare smallest elements of A and B and merge smaller into C.
- A = (2, 5, 6)
  B = (3, 8, 9, 10)
  C = (1)

Merge Two Sorted Lists

- A = (5, 6)
  B = (3, 8, 9, 10)
  C = (1, 2)
- A = (5, 6)
  B = (8, 9, 10)
  C = (1, 2, 3)
- A = (6)
  B = (8, 9, 10)
  C = (1, 2, 3, 5)

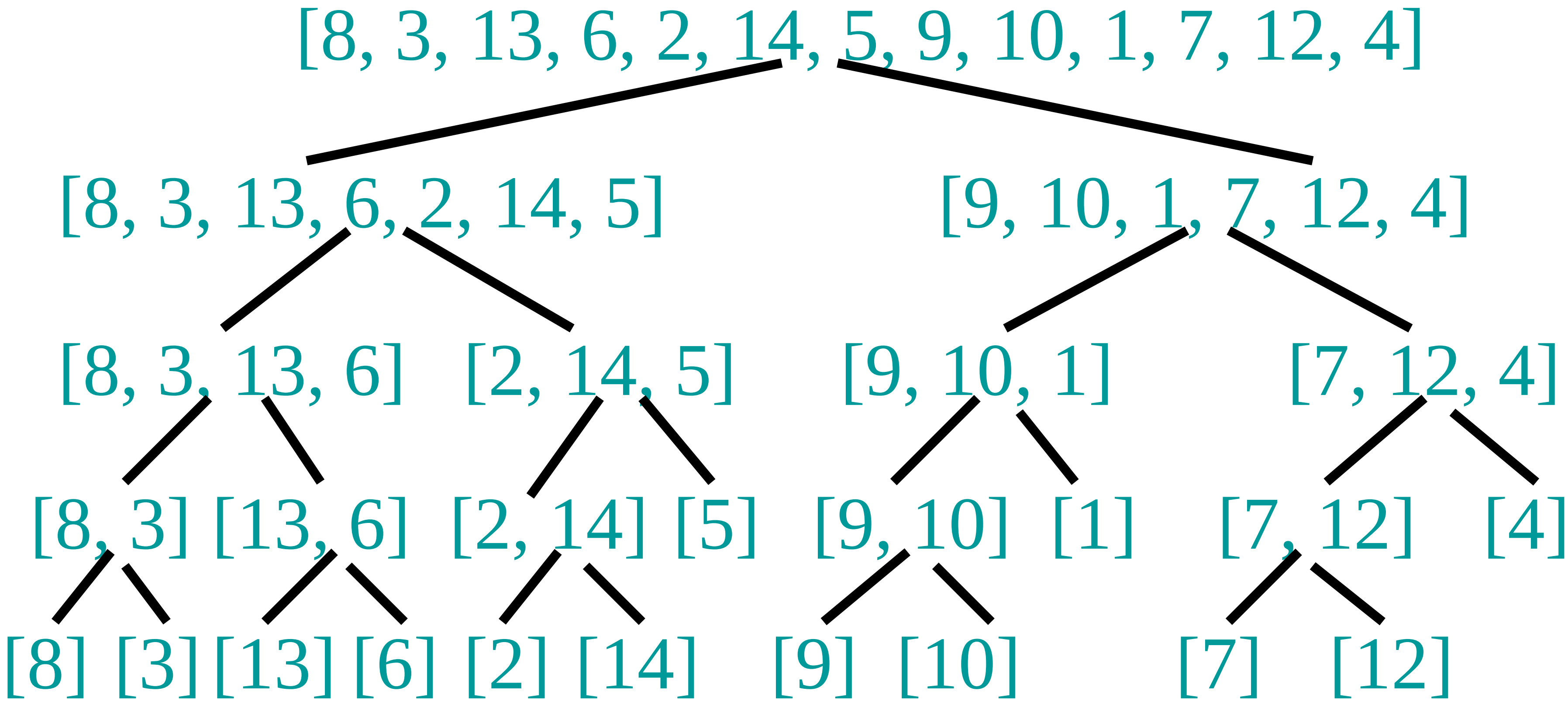
Merge Two Sorted Lists

- A = ()

  B = (8, 9, 10)

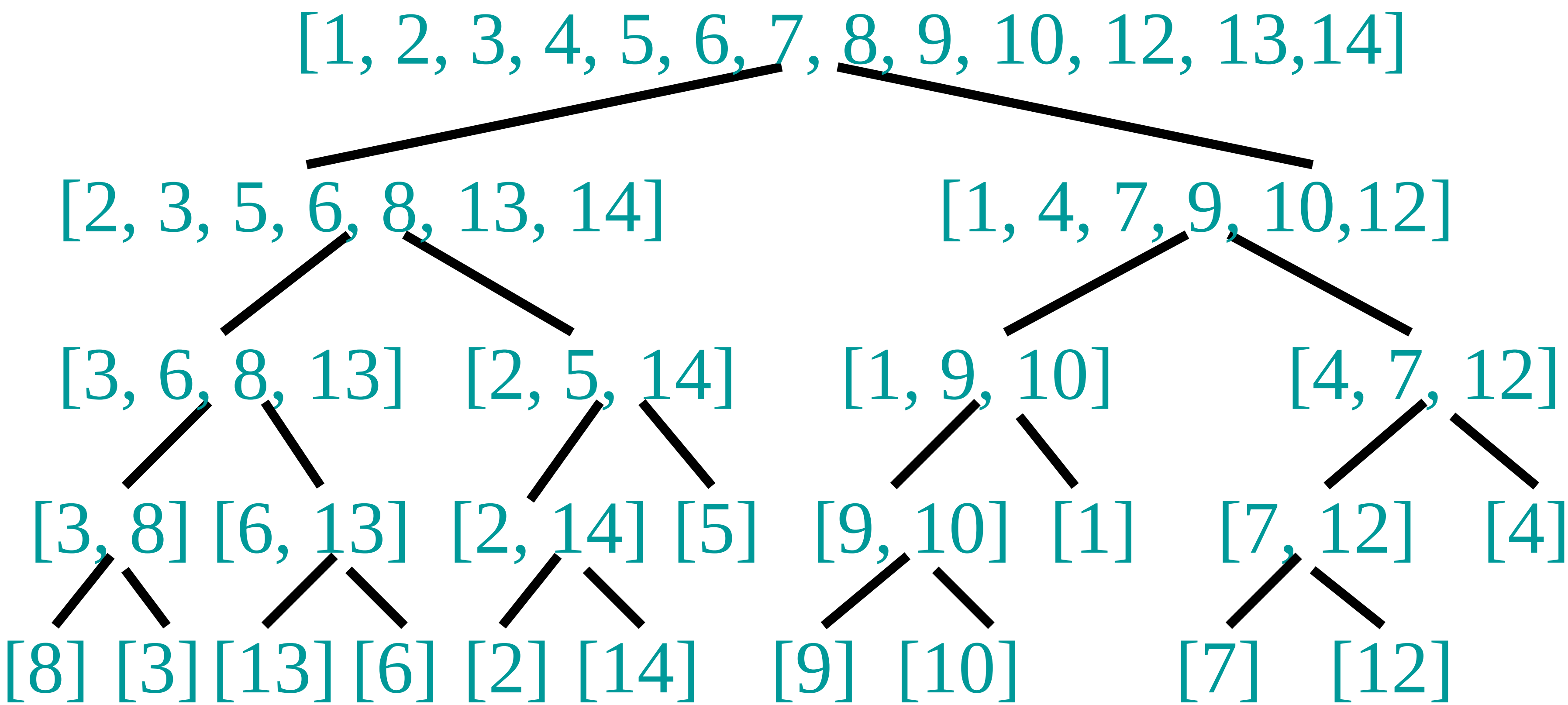
  C = (1, 2, 3, 5, 6)
- When one of A and B becomes empty, append the other list to C.
- $O(1)$ time needed to move an element into C.
- Total time is $O(n + m)$, where n and m are, respectively, the number of elements initially in A and B.

Merge Sort



[8, 3, 13, 6, 2, 14, 5, 9, 10, 1, 7, 12, 4]
[8, 3, 13, 6, 2, 14, 5]
[9, 10, 1, 7, 12, 4]
[8, 3, 13, 6]
[2, 14, 5]
[9, 10, 1]
[7, 12, 4]
[8, 3]
[13, 6]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Merge Sort


[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13,14]
[2, 3, 5, 6, 8, 13, 14]
[1, 4, 7, 9, 10,12]
[3, 6, 8, 13]
[2, 5, 14]
[1, 9, 10]
[4, 7, 12]
[3, 8]
[6, 13]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Time Complexity

- Let $t(n)$ be the time required to sort n elements.
- $t(0) = t(1) = c$, where c is a constant.
- When $n > 1$,

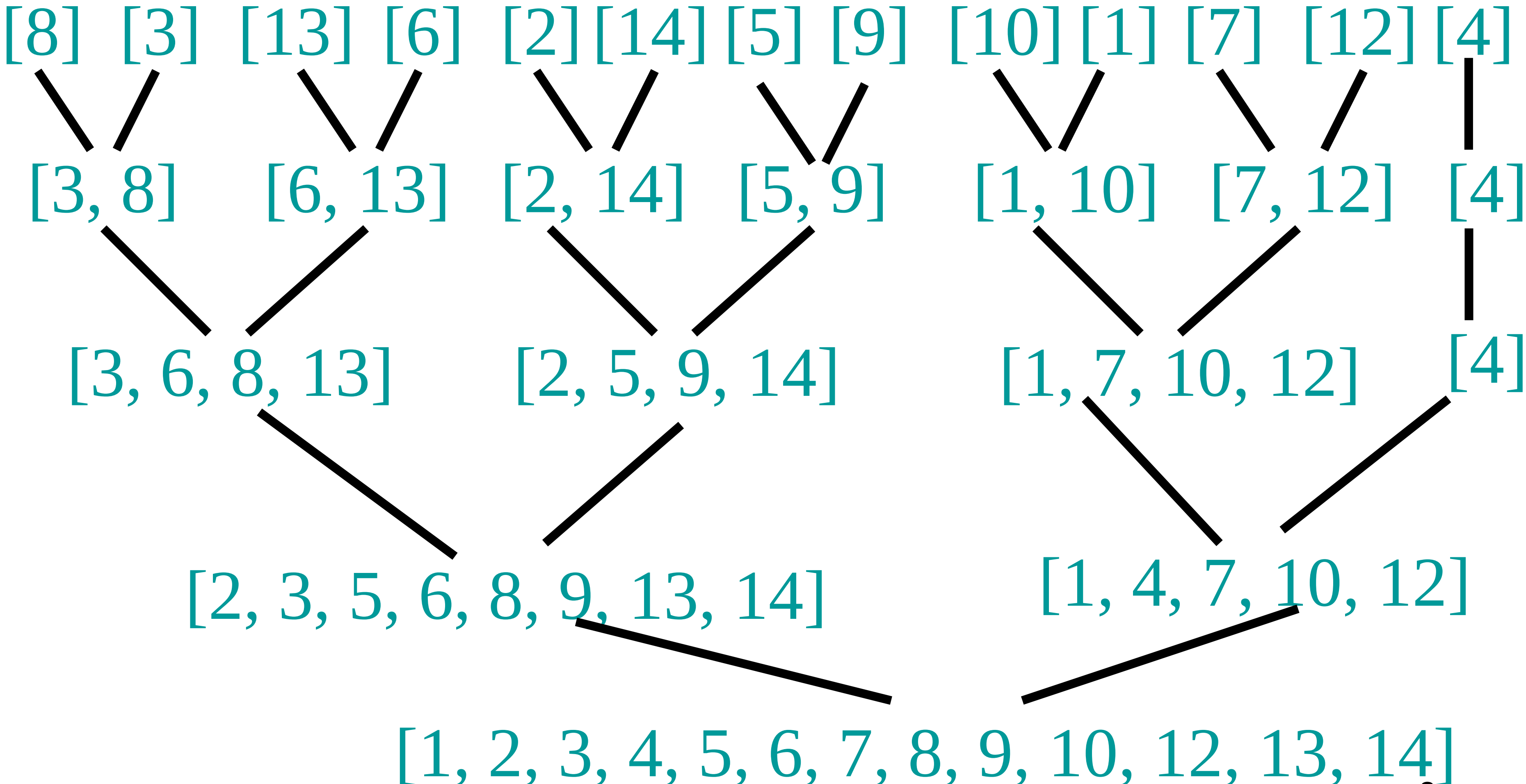
  $t(n) = t(ceil(n/2)) + t(floor(n/2)) + dn$,

  where d is a constant.
- To solve the recurrence, assume n is a power of 2 and use repeated substitution.
- $t(n) = O(n \log n)$.

Nonrecursive Version

- Eliminate downward pass.
- Start with sorted lists of size 1 and do pairwise merging of these sorted lists as in the upward pass.

Nonrecursive Merge Sort



Complexity

- Sorted segment size is 1, 2, 4, 8, …
- Number of merge passes is ceil($\log_2 n$).
- Each merge pass takes O(n) time.
- Total time is O(n log n).
- Need O(n) additional space for the merge.
- Merge sort is slower than insertion sort when n <= 15 (approximately). So define a small instance to be an instance with n <= 15.
- Sort small instances using insertion sort.
- Start with segment size = 15.

Natural Merge Sort

- Initial sorted segments are the naturally ocurring sorted segments in the input.
- Input = [8, 9, 10, 2, 5, 7, 9, 11, 13, 15, 6, 12, 14].
- Initial segments are:

  [8, 9, 10] [2, 5, 7, 9, 11, 13, 15] [6, 12, 14]
- 2 (instead of 4) merge passes suffice.
- Segment boundaries have a[i] > a[i+1].

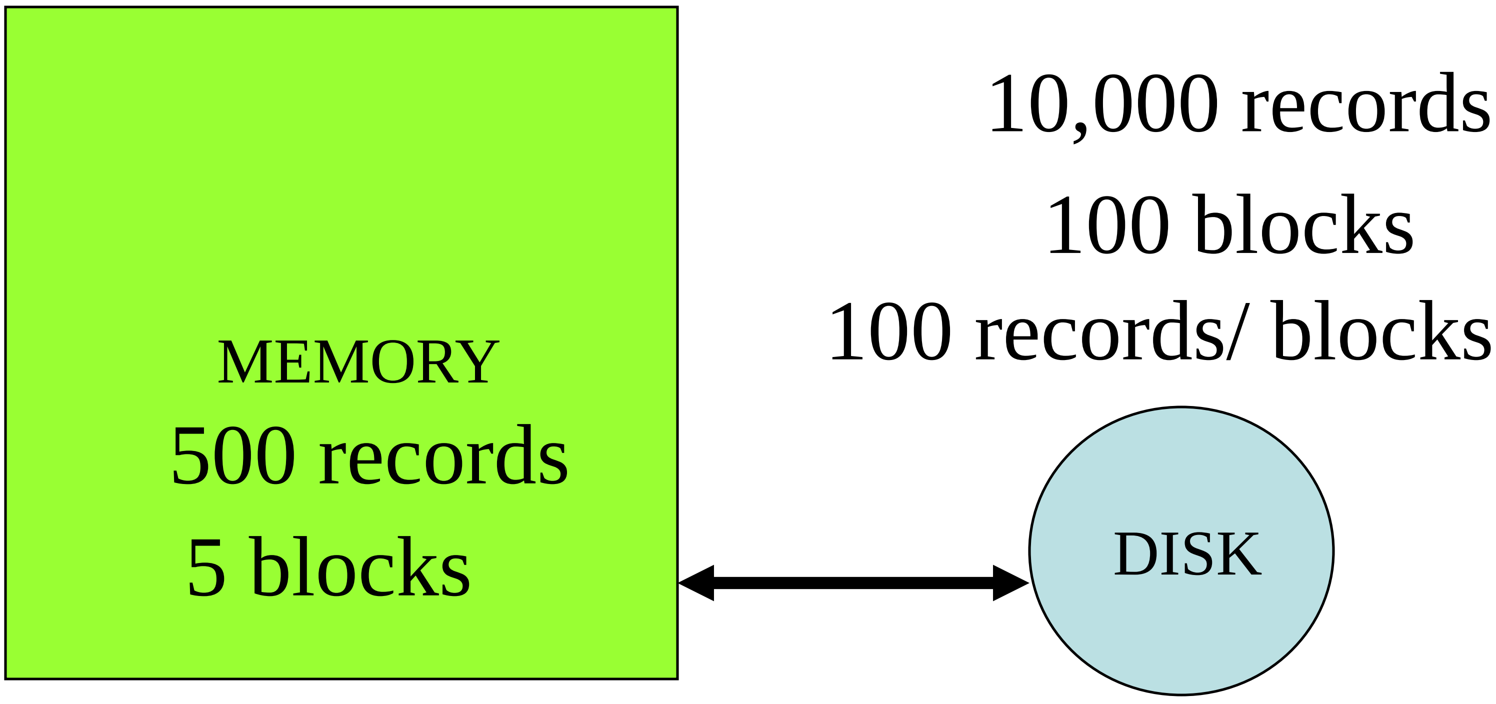
External Merge Sort

- 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 5 blocks as a run.
- Do 20 times.

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

Two-Way External Merge Sort

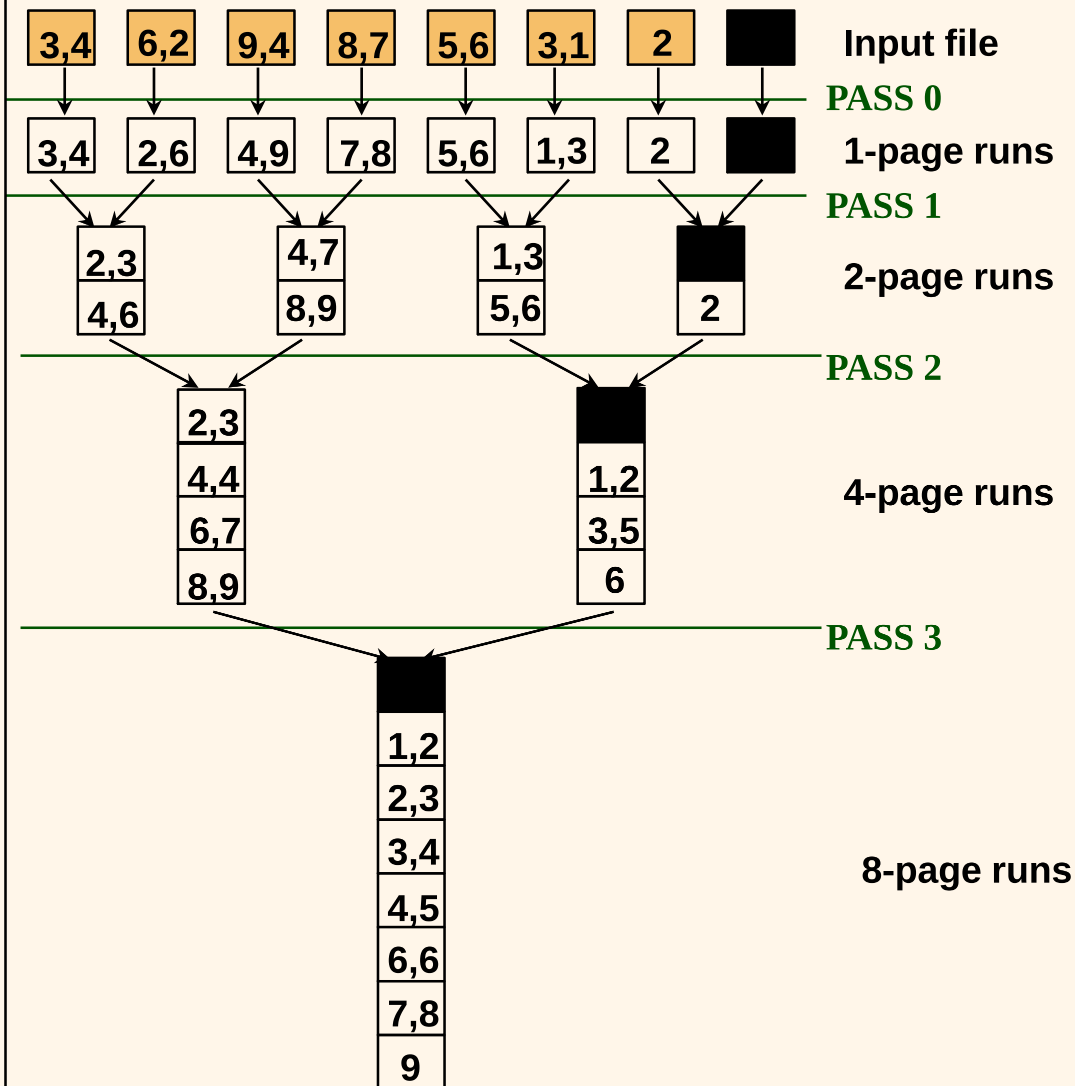
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

$$= \lceil \log_2 N \rceil + 1$$

- So toal cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

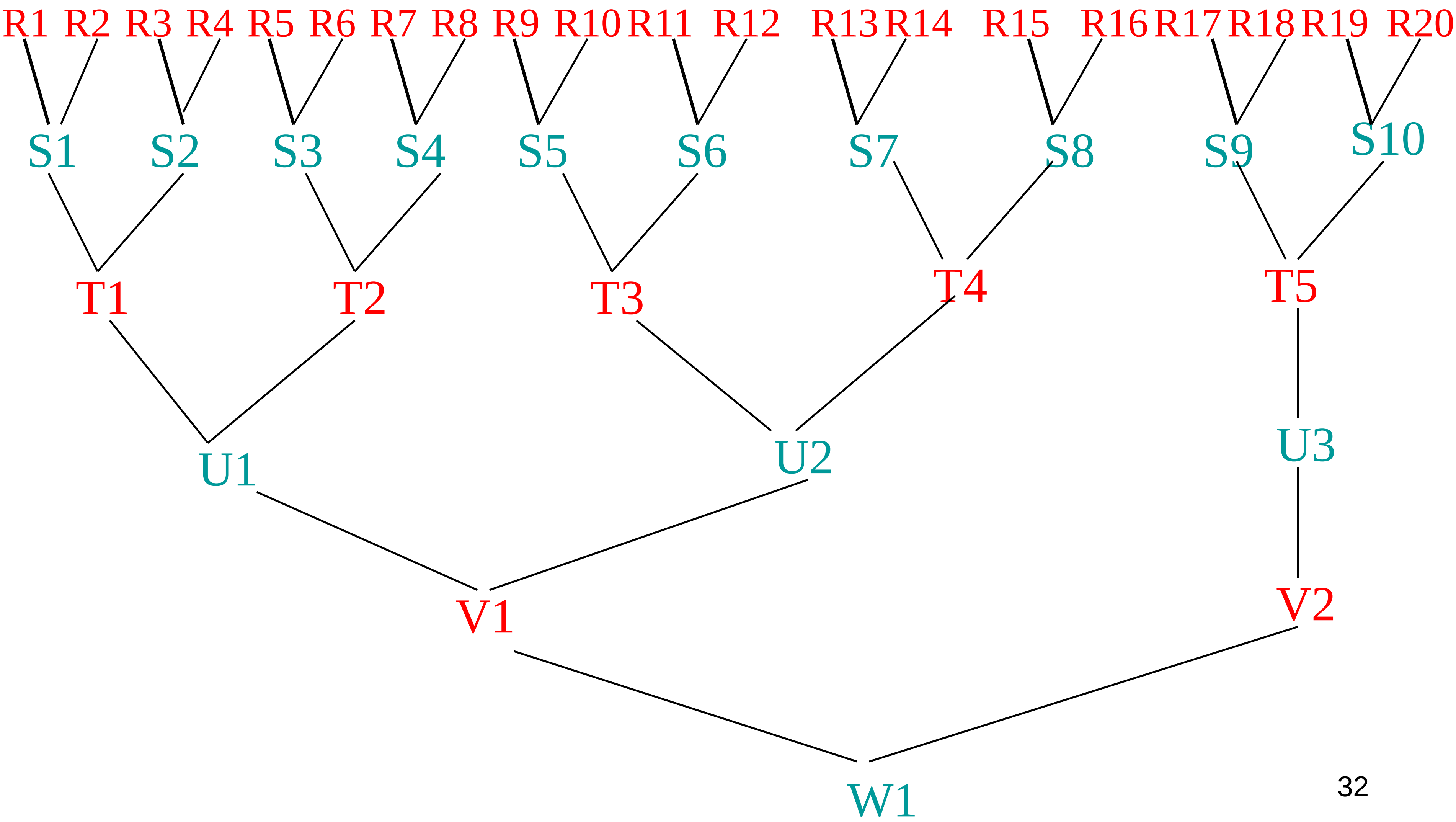
- <u>*Idea:*</u> ***Divide and conquer:*** sort subfiles and merge



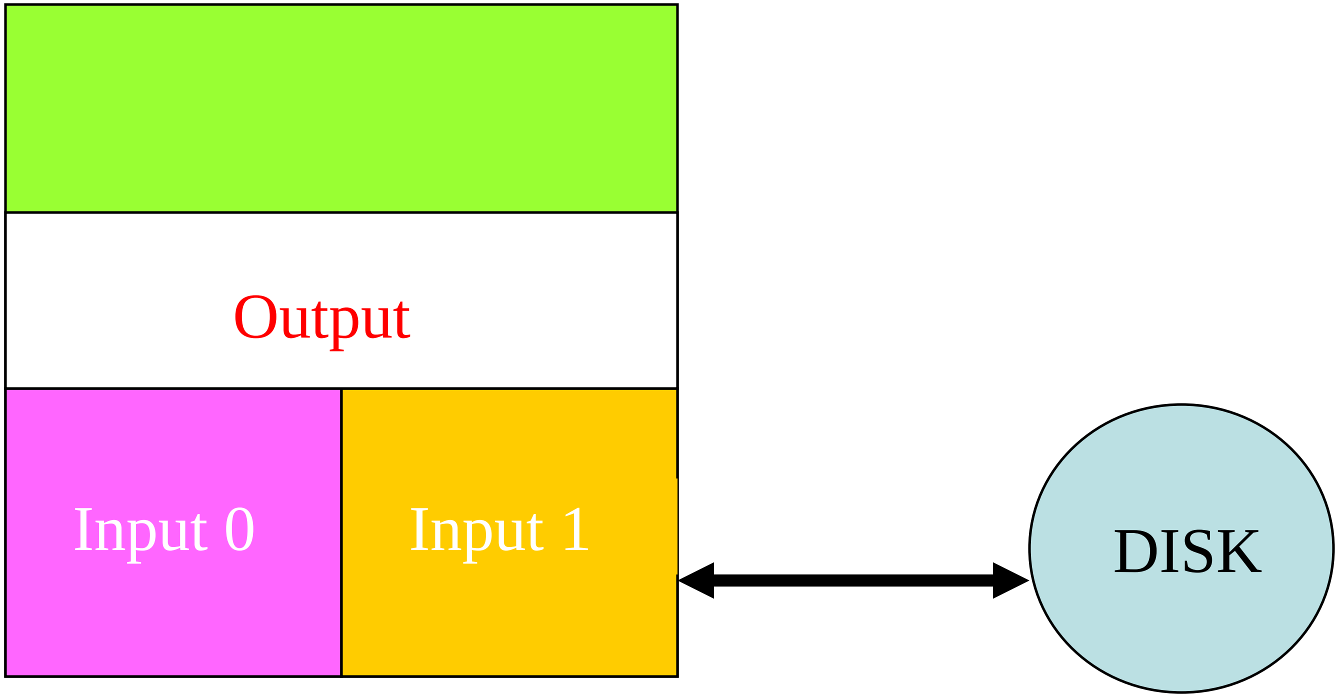
Run Merging

- 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 I_0 (Input 0) from R1 and I_1 from R2.
- Merge from I_0 and I_1 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))

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

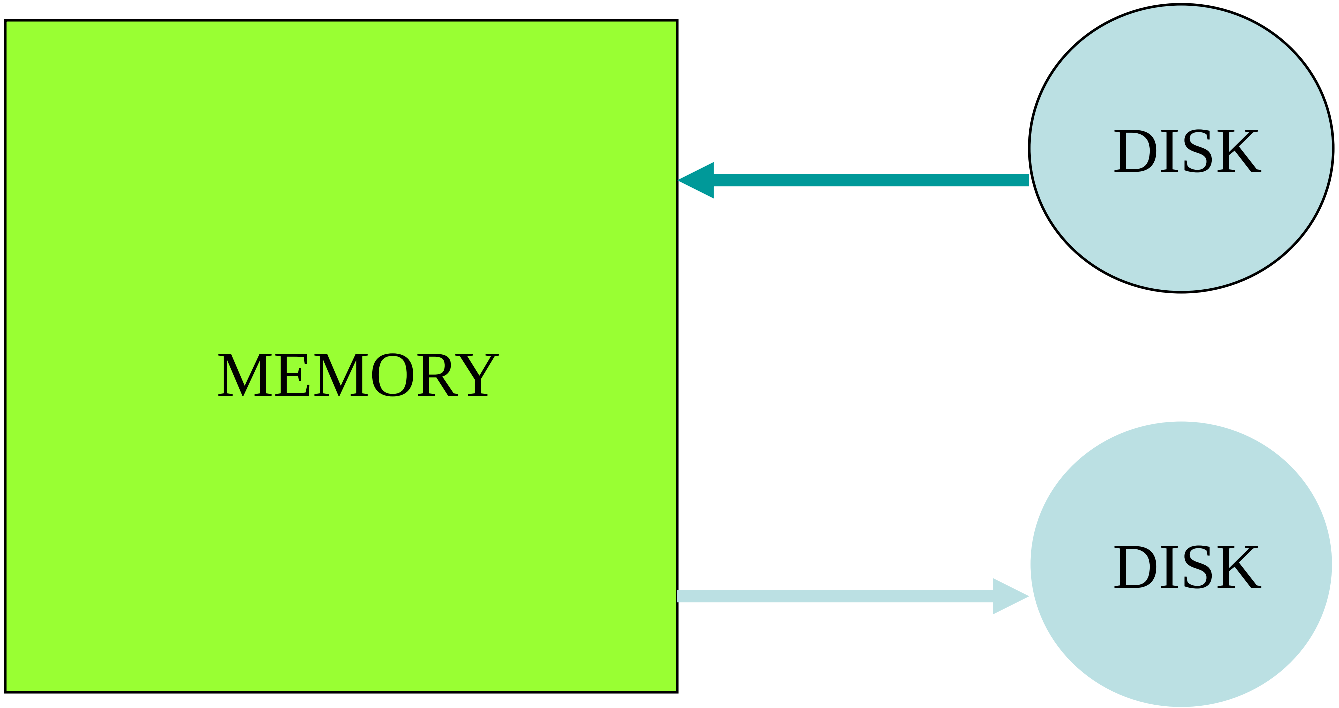
  = $(200t_{IO} + 100t_{IM}) * 5$

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.



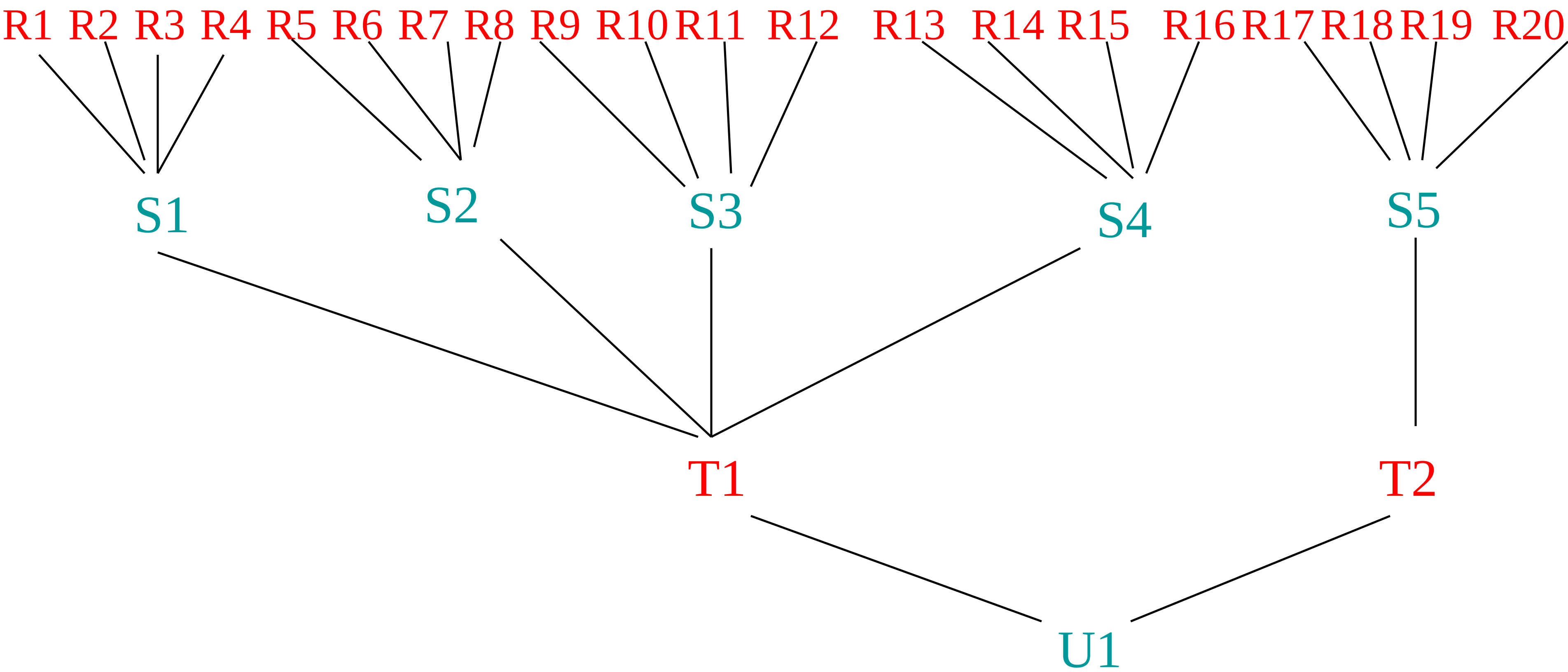
Improve Run Generation

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

Improve Run Merging

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

Merge 20 Runs Using 4-Way Merging



Number of merging passes = 3

Total passes = 1 (run generation) + 3 (mering) =4

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

- Time to merge 4-way of runs
  $40t_{IO} + 20t_{IM}$.
- Time to merge all runs (5 blocks/run)
  $200t_{IO} + 100t_{IM}$.

Total Run-Merging Time

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

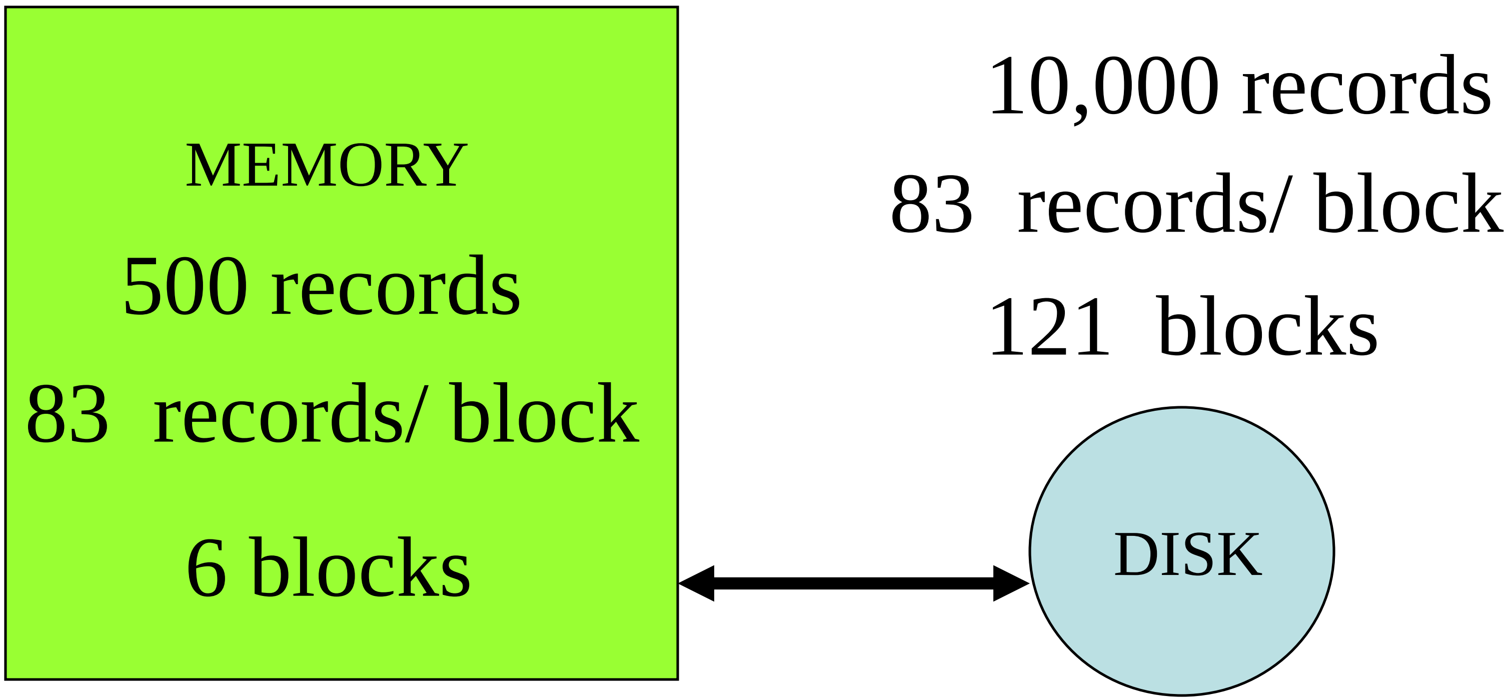
  = (time for one merge pass)

  $* \text{ceil}(\log_4(\text{number of initial runs}))$

  $= (200t_{IO} + 100t_{IM}) * \text{ceil}(\log_4(20))$

  $= (200t_{IO} + 100t_{IM}) * 3$

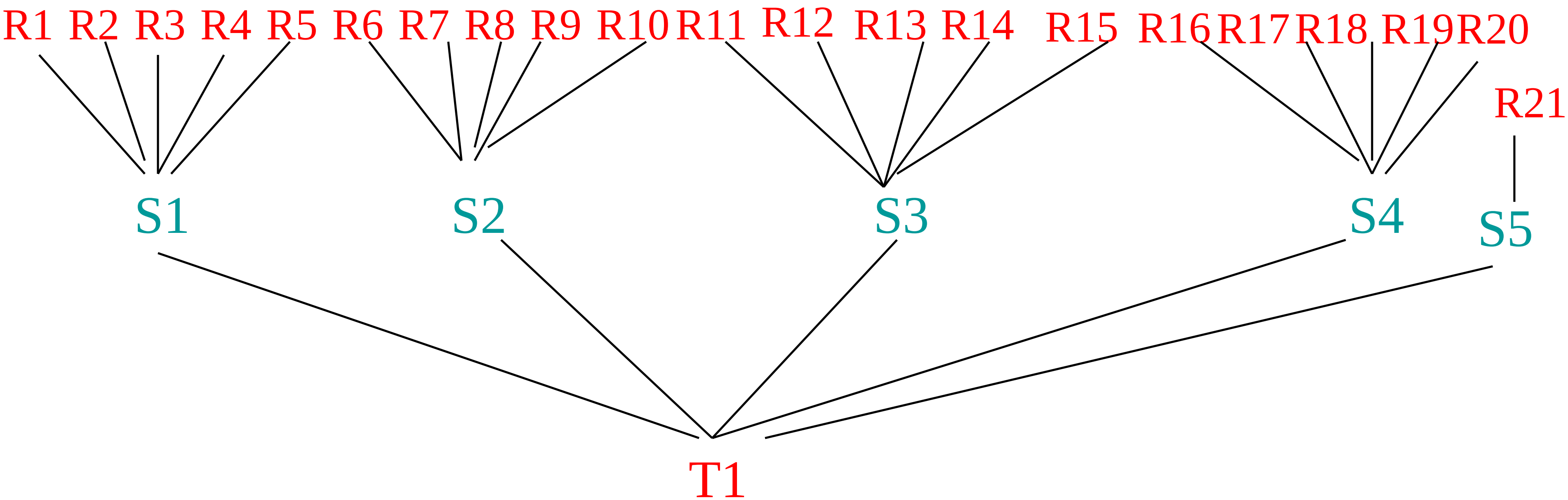
Run Generation



- Input 6 blocks.
- Sort.
- Output 6 blocks as a run.
- Do 21 times.

- 6 t_{IO}
- t_{IS}
- 6 t_{IO}
- $242t_{IO} + 21t_{IS}$

Merge 21 Runs Using 5-Way Merging



Number of merging passes = 2

Total passes = 1 (run generation)+2 (merging)

= 3

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

- Time to merge 5-way of runs
  $60t_{IO} + 30t_{IM}$.
- Time to merge all runs (6 blocks/run)
  $242t_{IO} + 121t_{IM}$.

Total Run-Merging Time

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

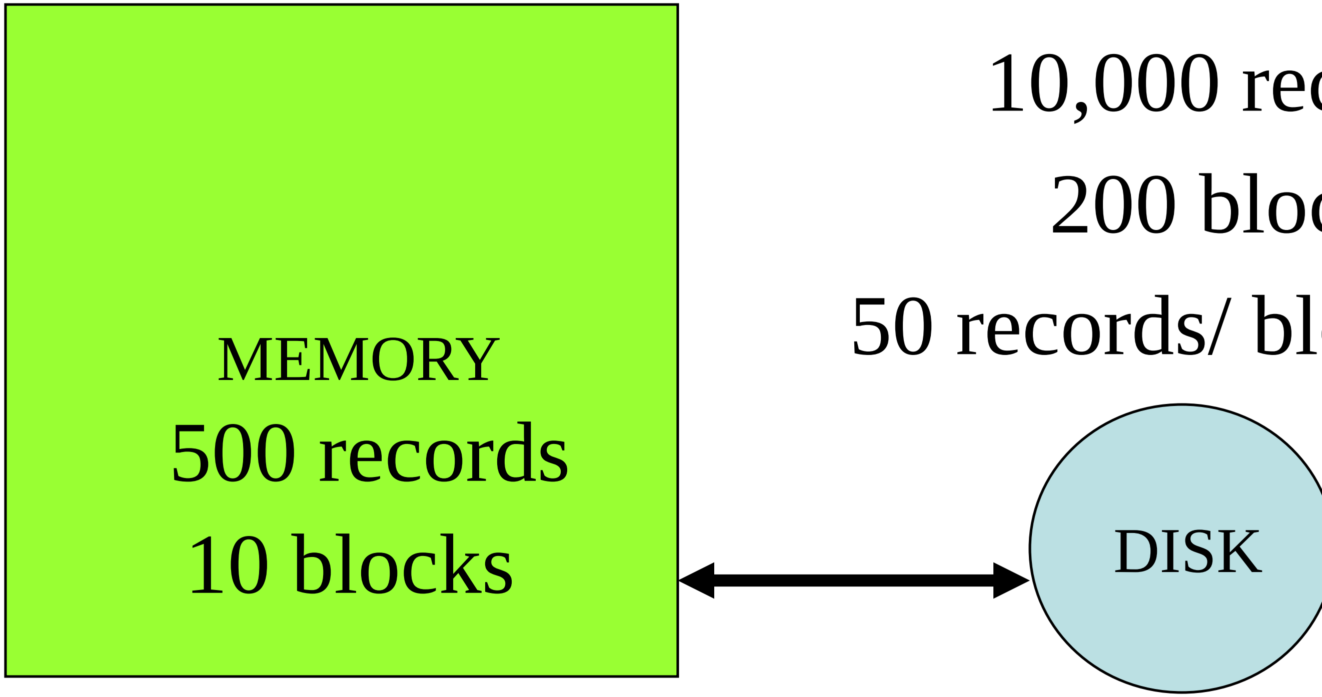
  = (time for one merge pass)

  * ceil($\log_5$(number of initial runs))

  = $(242t_{IO} + 121t_{IM}) * \text{ceil}(\log_5(21))$

  = $(242t_{IO} + 121t_{IM}) * 2$

Run Generation



10,000 records

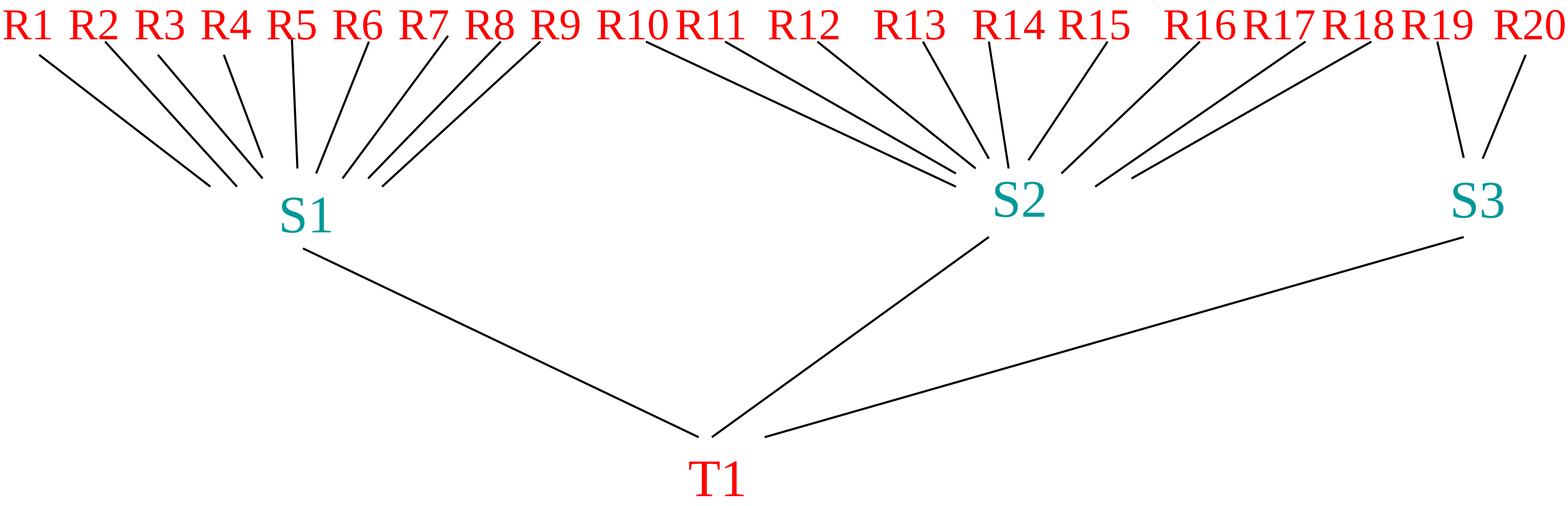
200 blocks

50 records/ blocks

- Input 10 blocks.
- Sort.
- Output 10 blocks as a run.
- Do 20 times.

- $10\ t_{IO}$
- t_{IS}
- $10\ t_{IO}$
- $400t_{IO} + 20t_{IS}$

Merge 20 Runs Using 9-Way Merging



Number of merging passes = 2

Total passes = 1 (run generation)+2 (merging)

= 3

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

- Time to merge 9-way of runs
  $180t_{IO} + 90t_{IM}$.
- Time to merge all runs
  $800t_{IO} + 400t_{IM}$.

Total Run-Merging Time

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

  = (time for one merge pass)

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

  = $(400t_{IO} + 200t_{IM}) * \text{ceil}(\log_9(20))$

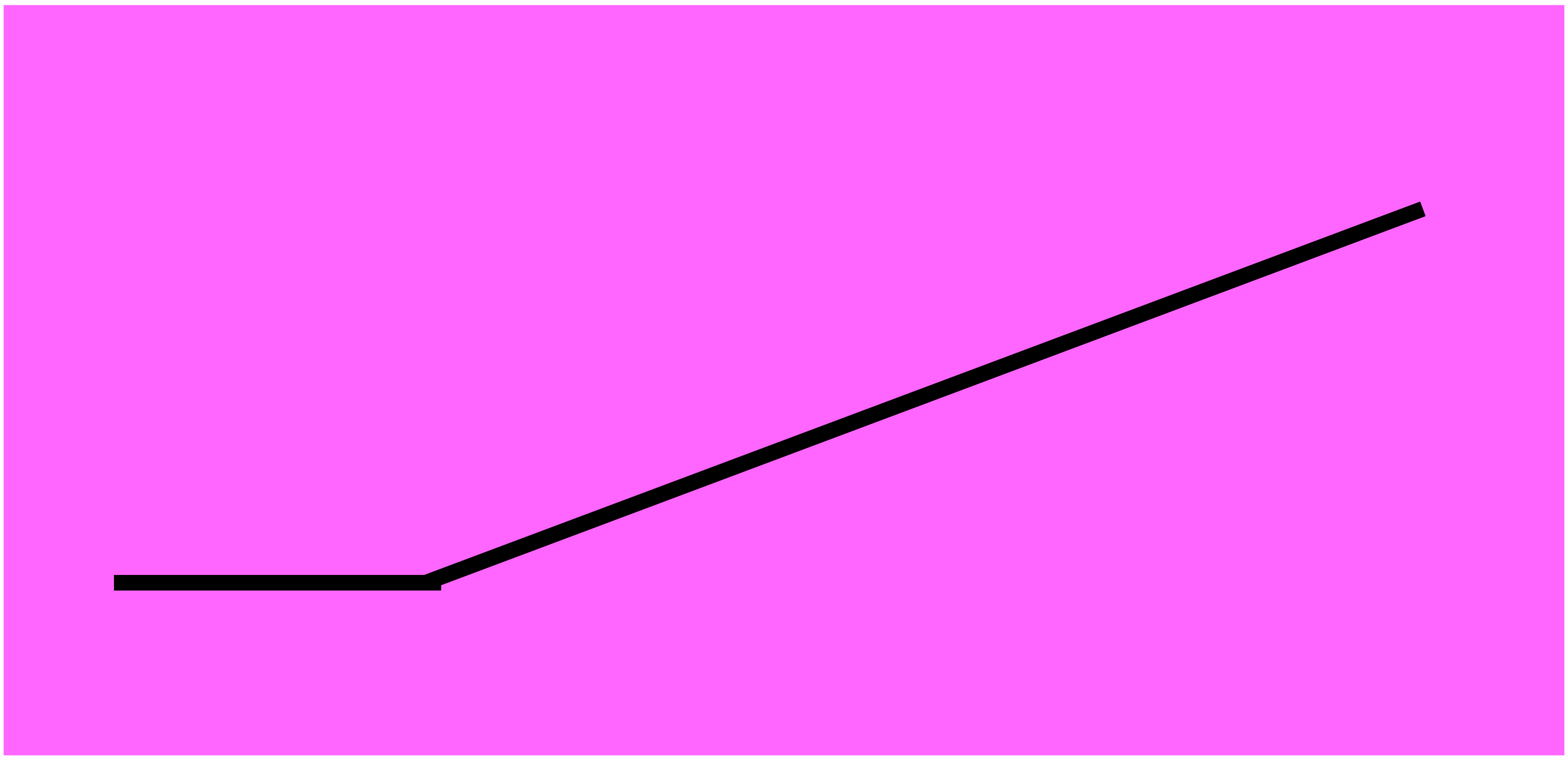
  = $(400t_{IO} + 200t_{IM}) * 2$

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

I/O time per pass

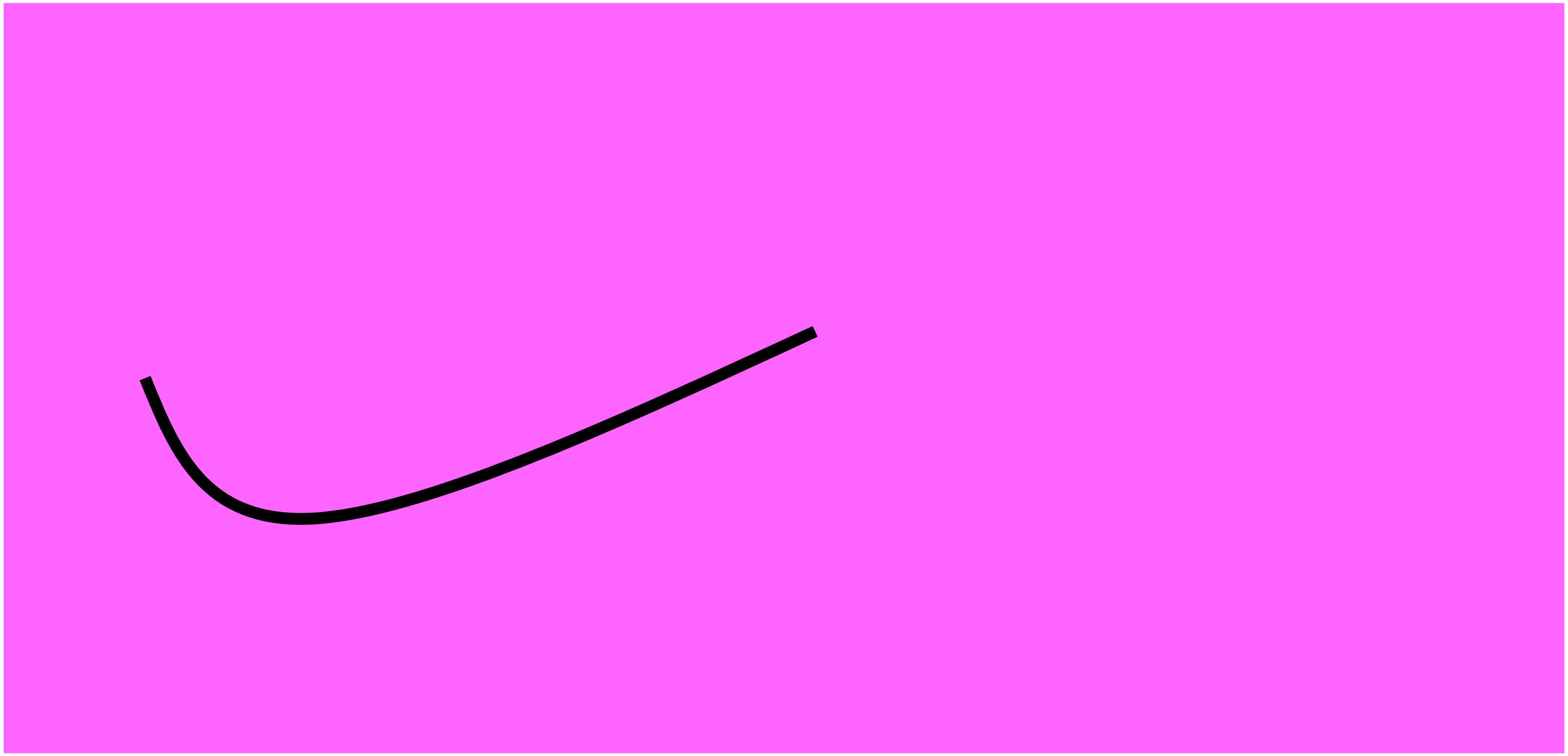


k

Total I/O Time To Merge Runs

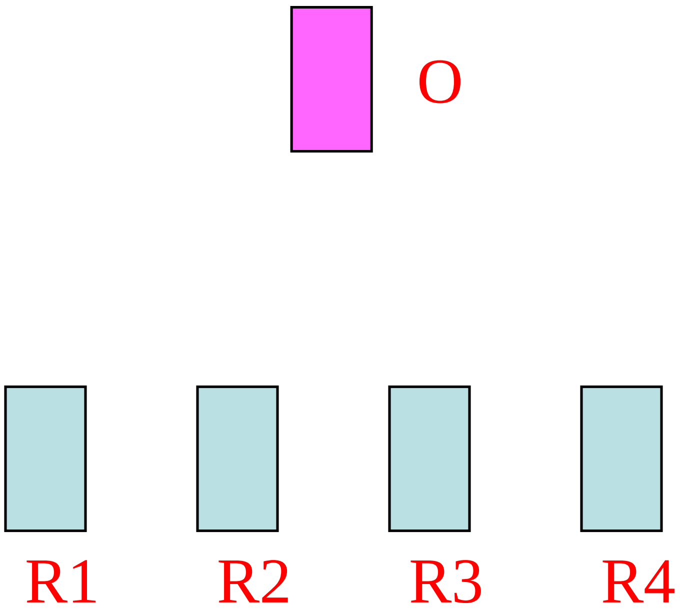
- (I/O time for one merge pass) * ceil($\log_k$(number of initial runs))

Total I/O time to merge runs



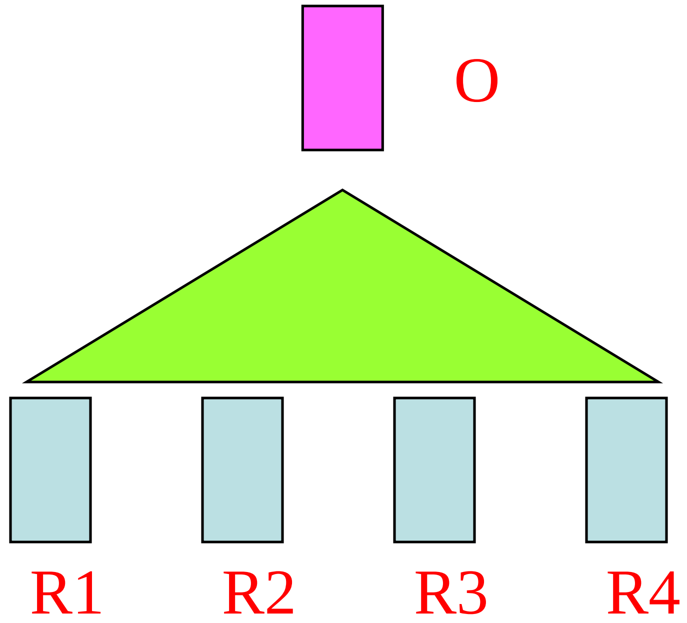
k

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 $c(k-1)n\log_k r$.

Merge Time Using A Selection 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 $(dn\log_2 k) \log_k r = dn\log_2 r$.

Improve Run Merging

- Reduce number of merge passes.
  - Use higher order merge.
  - Number of passes
    = ceil($\log_k$(number of initial runs))
    where k is the merge order.
- More generally, a higher-order merge reduces the cost of the optimal merge tree.