Handling large amount of data efficiently

1. Storage media and its constraints (magnetic disks, buffering)
2. Algorithms for large inputs (sorting)
3. Data structures (trees, hashes and bitmaps)
Algorithms for large inputs

External-memory sorting

By Marina Barsky
Winter 2016, University of Toronto
Algorithms for external memory

• In most studies of algorithms, one assumes the "RAM model":
  • Data is in main memory,
  • Access to any item of data takes as much time as any other.

• When implementing a DBMS, one must assume that the data does **NOT** fit into main memory.

• Often, the best algorithms for processing very large amounts of data differ from the best main-memory algorithms for the same problem.

• **There is a great advantage in choosing an algorithm that uses few disk accesses**, even if the algorithm is not very efficient when viewed as a main-memory algorithm.
I/O model of computation

- **Disk I/O** = read or write of a block is very expensive compared with what is likely to be done with the block once it arrives in main memory. Perhaps 1,000,000 machine instructions in the time to do one random disk I/O.

- The I/O model of computation measures the efficiency of an algorithm by counting how many disk reads and writes it needs.

- The unit of I/O is a **block**

- The model is oversimplified – no difference between consecutive reading of several blocks and random access


Merge Sort

• Common main-memory sorting algorithms don't look so good when you take disk I/O's into account. Variants of Merge Sort do better.

• Merge = take two sorted lists and repeatedly choose the smaller of the “heads” of the lists (head = first of the unchosen).
  • Example: merge 1,3,4,8 with 2,5,7,9 = 1,2,3,4,5,7,8,9.

• Merge Sort based on recursive algorithm:
  • divide records into two parts;
  • recursively mergesort the parts, and
  • merge the resulting lists.
Merge sort

algorithm mergesort (array A of size N)
    if ( N = 1 ) return A

    A1: = A[0 – N /2)
    A2: = A[N/2+1 – N)

    A1: = mergesort ( A1 )
    A2: = mergesort ( A2 )

    return merge (A1, A2)

merge (array X, array Y )
    result array Z
    i = 0  j = 0
    while ( i < |X| and j < |Y|)
        if ( X[i] > Y[j] )
            append Y[j] to Z
            j ++
        else
            append X[i] to Z
            i ++
    while (i < |X|)
        append X[i] to Z
        i ++
    while (j < |Y|)
        append Y[j] to Z
        j ++
2-way merge sort is not good enough for disk data

- If input is N blocks -
  - $\log_2 N$ passes - so each record is read/written from disk $\log_2 N$ times during merge.

- If data is on disk – $O(N \log N)$ disk I/Os
Two-Phase, **Multiway** Merge Sort

• The secondary-memory variant operates in a small number of *passes*;
  • in each pass every record is read into main memory once and written out to disk once.

• **2PMMS**: 2 reads + 2 writes per block.
2PMMS: Phase 1

1. Fill main memory with records.
2. Sort using favorite main-memory sort.
3. Write sorted sublist to disk.
4. Repeat until all records have been put into one of the sorted lists (*runs*).
2PMMS: Phase 2

- Manage the buffers as needed:
  - If an input block is exhausted, get the next block from the same run.
  - If the output block is full, write it to disk.
2PMMS: Toy Example

• 24 records with keys:
  12 10 25 20 40 30 27 29 14 18 45 23 70 65 35 11 49 47 22 21 46 34 29 39

• Suppose 1 block can hold 2 records.
• Suppose main memory (MM) can hold 4 blocks i.e. 8 records.

Phase 1.

• Load 12 10 25 20 40 30 27 29 in MM, sort them and write the sorted sublist: 10 12 20 25 27 29 30 40
• Load 14 18 45 23 70 65 35 11 in MM, sort them and write the sorted sublist: 11 14 18 23 35 45 65 70
• Load 49 47 22 21 46 34 29 39 in MM, sort them and write the sorted sublist: 21 22 29 34 39 46 47 49
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: □ □ □
Input Buffer2: □ □ □
Input Buffer3: □ □ □
Output Buffer: □ □ □

Sorted list:
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 10 12
Input Buffer2: 11 14
Input Buffer3: 21 22
Output Buffer: 

Sorted list:
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 10 12
Input Buffer2: 11 14
Input Buffer3: 21 22
Output Buffer: 10

Sorted list:
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 10 12
Input Buffer2: 11 14
Input Buffer3: 21 22
Output Buffer: 10 11

Sorted list:
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1:  
Input Buffer2:  
Input Buffer3:  
Output Buffer:  

Sorted list:

Output Buffer full: flush to disk
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 10 12
Input Buffer2: 11 14
Input Buffer3: 21 22
Output Buffer: 

Sorted list: 10 11
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 
\[
\begin{array}{|c|c|}
\hline
10 & 12 \\
\hline
\end{array}
\]

Input Buffer2: 
\[
\begin{array}{|c|c|}
\hline
11 & 14 \\
\hline
\end{array}
\]

Input Buffer3: 
\[
\begin{array}{|c|c|}
\hline
21 & 22 \\
\hline
\end{array}
\]

Output Buffer: 
\[
\begin{array}{|c|}
\hline
12 \\
\hline
\end{array}
\]

Sorted list: 10 11

Processed Input Buffer1 – upload from sub-list 1
2PMMS example – Phase II

Phase 2.

On disk:
- Sub-list 1: 10 12 20 25 27 29 30 40
- Sub-list 2: 11 14 18 23 35 45 65 70
- Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)
- Input Buffer1: 20 25
- Input Buffer2: 11 14
- Input Buffer3: 21 22
- Output Buffer: 12

Sorted list: 10 11
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 20 25
Input Buffer2: 11 14
Input Buffer3: 21 22
Output Buffer: 12 14

Processed Input Buffer2 – upload from sub-list 2

Output Buffer full: flush to disk

Sorted list: 10 11
2PMMS example – Phase II

Phase 2.

On disk:

Sub-list 1: 10 12 20 25 27 29 30 40
Sub-list 2: 11 14 18 23 35 45 65 70
Sub-list 3: 21 22 29 34 39 46 47 49

Main Memory (4 buffers)

Input Buffer1: 20 25
Input Buffer2: 18 23
Input Buffer3: 21 22
Output Buffer: 

We continue in this way until the sorted sub-lists are finished and we get on disk the whole sorted list of records.

Sorted list: 10 11 12 14 …
Merge sort: how long does it take?

- **10,000,000** records of **200** bytes each = **2GB** file.
  - Stored on disk, with **20K** blocks, each holding **100** records
  - Entire file occupies **100,000** blocks \(\frac{200 \times 10^7}{20 \times 10^3}\)

- **100MB** available **main memory**
  - The number of blocks that can fit in 100MB of memory is \(100 \times 10^6 \div (20 \times 10^3)\), or \(\approx 5,000\) blocks.

**I/O time per block – avg 11.5 ms:**
- 6 ms – average seek time
- 5 ms – average rotational delay
- 0.5 ms – transfer time for 20K block
Analysis – Phase 1

• **5000** out of the **100,000** blocks will fill main memory.

• We thus fill memory $\left\lfloor \frac{100,000}{5,000} \right\rfloor = 20$ times, sort the records in main memory, and write 20 sorted runs out to disk.

• How long does this phase take?

• We read each of the **100,000** blocks once, and we write **100,000** new blocks. Thus, there are **200,000** disk I/O's for $200,000 \times 11.5$ ms = 2300 seconds, or **38 minutes**.

I/O time per block – avg 11.5 ms:
6 ms – average seek time
5 ms – average rotational delay
0.5 ms – transfer time for 20K block
Analysis – Phase 2

• Every block holding records from one of the sorted lists is read from disk exactly once.
  • Thus, the total number of block reads is 100,000 in the second phase, just as for the first.
• Likewise, each record is placed once in an output block, and each of these blocks is written to disk once.
  • Thus, the number of block writes in the second phase is also 100,000.

• We conclude that the second phase takes another 38 minutes.
• Total: Phase 1 + Phase 2 = 76 minutes.
Advantage of larger blocks

• In our analysis block size is 20K.
• What would happen if the block size is 40K?

• The number of blocks to read and write is now 50,000 (half of 100,000)

• Each read and write takes longer, so each I/O is more expensive.
• Now, the only change in the time to access a block would be that the transfer time increases to 0.50*2=1 ms, the average seek time and rotational delay remain the same.
• The time for block size 40K: (2 * 50,000 disk I/Os for phase I + 2 * 50,000 disk I/Os for phase II )*12 = 2400 ms for both phases = 40 min (vs. 76)

I/O time per block – avg 12 ms:
6 ms – average seek time
5 ms – average rotational delay
1 ms – transfer time for 40K block
Another example: block size = 500K

- For a block size of 500K the transfer time per block is 0.5*25=12.5 milliseconds.
- Average block access time would be 11 + 12.5 approx. 24 ms, (as opposed to 11ms we had)
- However, now a block can hold 100*25 = 2500 records and the whole table will occupy 10,000,000 / 2500 = 4000 blocks (as opposed to 100,000 blocks we had before).
- Thus, we would need only 4000 * 2 * 2 disk I/Os for 2PMMS for a total time of 4000 * 2 * 2 * 24 = 384,000ms or about 6.4 min.
- Speedup: 76 / 6.4 = 12 fold !
Reasons to limit the block size

1. We cannot use blocks that cover several tracks effectively.

2. Small relations would occupy only a fraction of a block, so large blocks would waste space on disk.

3. The larger the blocks are, the fewer records we can sort by 2PMMS (less runs we can merge in Phase II).

```bash
$ sudo hdparm -I /dev/sda
/dev/sda:
ATA device, with non-removable media
  Model Number:       ST3500630AS
  Serial Number:      9XXYZ845YZ
  Firmware Revision:  3.AAK
Standards:
  Supported: 7 6 5 4
  Likely used: 7
Configuration:
  Logical max current
cylinders 16383 16383
  heads 16 16
  sectors/track 63 63
  --
  CHS current addressable sectors:
```

Nevertheless, as machines get more memory and disks more capacious, there is a tendency for block sizes to grow.
Max number of records we can sort in 2 passes

Phase 2

• How many input buffers we can have at most in Phase II?

\[
\frac{M}{B} - 1
\]

at least 1 block per buffer
one separate block for output buffer

Phase 1

• How many sorted sublists max?

\[(\frac{M}{B}) - 1\]

• How many records max we can sort in 1 sublist?

\[
\frac{M}{R}
\]

Hence, we are able to sort \((\frac{M}{R})^*[(\frac{M}{B})-1]\) \(\approx\) \(\frac{M^2}{RB}\) records
Max number of records we can sort in 2 passes: Example

\[ M = 100\text{MB} = 100,000,000 \text{ Bytes} = 10^8 \text{ Bytes} \]
\[ B = 20,000 \text{ Bytes} \]
\[ R = 100 \text{ Bytes} \]

So, \[ \frac{M^2}{RB} = \frac{(10^8)^2}{100 \times 20,000} = 6 \times 10^9 \text{ records}, \]
100 bytes each

or relation of 600 GB
- just with 100MB of memory!

- block size in bytes.
- main memory in bytes.
- size of one record in bytes.
How many runs to have

- In general, we want to have a larger buffer for each run where we would buffer several blocks at a time, taking advantage of sequential access, and thus we want to have a comparatively small number of big runs $k$:

- Phase I: $k > \frac{N}{M}$, because we can sort at most $M$ bytes of input in RAM

- Phase II: $k < \frac{M}{B} - 1$, because we cannot have more input buffers of size at least 1 block

\[
\frac{M}{B} - 1 > k > \frac{N}{M}
\]

Space for experiments: Assignment 2
Sorting larger relations

\[ \frac{M}{B} - 1 > k > \frac{N}{M} \]

- What if \( \frac{N}{M} > \frac{M}{B} - 1 \)?
  
i.e. we need to produce so many runs in Phase I, that we cannot allocate at least 1 block for each run in Phase II?

- Then we first produce much longer runs using 2PMMS: each run will be of max length \( \frac{M}{B} \times M \)

- In Phase III we merge this runs, the maximum we can merge is \( \frac{M}{B} \) runs of size \( \frac{M^2}{B} \) each.

- With 3 passes we can sort relation of size \( \frac{M^3}{B^2} \) bytes!
Max size we can sort in 3 passes

\[ M^3 / B^2 \]

- Memory \( M = 100 \) MB = \( 10^8 \) bytes
- Block size = \( 20 \) KB = \( 2\times10^4 \) bytes

- In 3 passes we can sort

\[
(10^8)^3 / 4 \times 10^8 = 2.5 \times 10^{15} \text{ bytes} = 2.5 \text{ PB!}
\]
General complexity of multi-way merge sort in I/O model

- We can sort a relation of any size $N$ with given amount of main memory $M$ and block size $B$ in $p$ passes.
- The total number of disk I/Os would be $4\cdot p\cdot N/B$
  $$N = M^p/B^{p-1}$$

- To find $p$, take $\log_B$ of both sides
  $$\log_B N = p \cdot \log_B M - (p-1) \log_B B$$
  $$p \approx \log_B N / \log_B M = \log_M N$$

- So the overall complexity in I/O model is $O(N \log_M N)$ disk I/Os.
Sorting efficiency is crucial:

- Duplicate elimination
- Grouping and aggregation
- Union, intersection, difference
- Join
Improving the Running Time of 2PMMS

• Blocked I/O
  Reading into buffer $P$ blocks (pages) at a time, instead of one block

• Double-buffering ("Prefetching")
  2 block-buffers per run, once the first is processed, it gets refilled while CPU is working on the second

• Cylindrification and multiple disks
Cylindrification

**Idea:** Place all input blocks by cylinder, so once we reach that cylinder we can read block after block, with no seek time or rotational latency (Assuming this is the only process running in the machine that needs disk access).

**Application to Phase 1 of 2PMMS**

- Because we have only transfer time, we can do Phase I for sorting 100,000 blocks (2 I/Os – read and write) in:
  
  \[
  2 \times 100,000 \times 1 \text{ms} = 200 \text{ sec} = 3.3 \text{ min}
  \]

**But, Phase 2 ...?**

- I/O time per block – avg 12 ms:
  - 6 ms – average seek time
  - 5 ms – average rotational delay
  - 1 ms – transfer time for 40K block
- I/O time per block – 1 ms:
Cylindrification – Phase 2

• Storage by cylinders does not help in Phase II
  • Blocks are read from the fronts of the sorted lists in an order that is determined by which list next exhausts its current block.
  • Output blocks are written one at a time, interspersed with block reads

• Thus, the second phase will still take 38 min.
• Total: 38+3 = 41 min vs 76 min

• We have cut the sorting time almost half, but cannot do better by cylindrification alone.
Multiple Disks and Cylindrification

• Use several disks with independent heads

• **Example**: Instead of a large disk of 1TB, let’s use 4 smaller disks of 250GB each
  • We divide the given records among the four disks; the data will occupy adjacent cylinders on each disk.
  • We distribute each sorted list onto the four disks, occupying several adjacent cylinders on each disk.

```
1 2 3 4
5 6 7 8
```

```
1 2 3 4
5 6 7 8
```

```
1 2 3 4
5 6 7 8
```
Multiple Disks – Phase 2

• Phase 2:
  • Use 4 output buffers, one per disk, and cut writing time in about 1/4.
  
  • When we need to fill an input buffer, we can perform several block reads in parallel. Potential for a 4-fold speedup.
  • In practice, we get 2-3 fold speedup for Phase 2.

• Total time ≈ 3 + 38/3 ≈ 15 min
Pathologies of Big Data

Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.
Improvement over the tape...

Jacobs:

“A further point that’s widely under-appreciated: in modern systems, as demonstrated in the figure, random access to memory is typically slower than sequential access to disk. Note that random reads from disk are more than 150,000 times slower than sequential access; SSD improves on this ratio by less than one order of magnitude.

In a very real sense, all of the modern forms of storage improve only in degree, not in their essential nature, upon that most venerable and sequential of storage media: the tape.”
A Tape algorithm (Knuth, Vol. 3)

• Balanced 2-way merge with 4 "working tapes"
  • During the first phase, sorted runs produced by the main-memory sorting are placed alternately on Tapes 1 and 2,
  • Then Tapes 1 and 2 are *rewound* to their beginnings, and we merge the runs from these tapes, obtaining new runs which are twice as long as the original ones;
  • The new runs are written alternately on Tapes 3 and 4 as they are being formed.
  • Then all tapes are rewound, and the contents of Tapes 3 and 4 are merged into quadruple-length runs recorded alternately on Tapes 1 and 2.
A Tape algorithm (cont.)

• The process continues, doubling the length of runs each time, until only one run is left (namely the entire sorted file).

• If $S$ runs were produced during the internal sorting phase this balanced 2-way merge procedure makes $\log S$ merging passes over all the data (not $\log N$)

<table>
<thead>
<tr>
<th>Pass</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A_1A_1A_1$</td>
<td>$A_1A_1A_1$</td>
<td>—</td>
<td>—</td>
<td>Initial distribution</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>$D_2D_2$</td>
<td>$D_2D_2$</td>
<td>Merge to T3 and T4</td>
</tr>
<tr>
<td>3</td>
<td>$A_4$</td>
<td>$A_4$</td>
<td>—</td>
<td>—</td>
<td>Merge to T1 and T2</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>$D_8$</td>
<td>—</td>
<td>Final merge to T3</td>
</tr>
</tbody>
</table>

From Knuth, “The art of computer programming”, Vol 3, p.299
Tape algorithm example. Phase I

- 5000 records are to be sorted with a main memory capacity of 1000

\[
\begin{align*}
&\text{Sort } R_1 \ldots R_{1000}; R_{1001} \ldots R_{2000}; R_{2001} \ldots R_{3000}; R_{3001} \ldots R_{4000} \\
&\quad R_{4001} \ldots R_{5000}\text{ and distribute into tape 1 and tape 2:}
\end{align*}
\]

Tape 1: \( R_1 \ldots R_{1000}; R_{2001} \ldots R_{3000}; R_{4001} \ldots R_{5000} \)
Tape 2: \( R_{1001} \ldots R_{2000}; R_{3001} \ldots R_{4000} \)
Tape 3: (empty)
Tape 4: (empty)
Tape algorithm example. Phase II

• Input to Phase II:
  Tape 1: $R_1 \ldots R_{1000}; R_{2001} \ldots R_{3000}; R_{4001} \ldots R_{5000}$
  Tape 2: $R_{1001} \ldots R_{2000}; R_{3001} \ldots R_{4000}$
  Tape 3: (empty)
  Tape 4: (empty)

• Output of Phase II:
  Tape 3: $R_1 \ldots R_{2000}; R_{4001} \ldots R_{5000}$
  Tape 4: $R_{2001} \ldots R_{4000}$

Tape 1: to erase
Tape 2: to erase
Tape algorithm example. Phase III

- Input to Phase III:
  Tape 3: $R_1 \ldots R_{2000}; R_{4001} \ldots R_{5000}$
  Tape 4: $R_{2001} \ldots R_{4000}$

- Output of Phase II:
  Tape 1: $R_1 \ldots R_{4000}$
  Tape 2: $R_{4001} \ldots R_{5000}$

- Finally – merge Tape 1 and Tape 2
Tape algorithm: toy example

- Sorted runs are written alternately to tapes 1 and 2:
  \[1, 4, 6, 3, 8, 10, 2, 7, 11, 5, 9, 12\]

Tape 1: 
\[1, 4, 6, 2, 7, 11\]

Tape 2: 
\[3, 8, 10, 5, 9, 12\]

Empty tapes:
- Tape 3:
- Tape 4:
Tape algorithm: toy example – rewind 2 tapes

Tape 1: 1, 4, 6, 2, 7, 11
Tape 2: 3, 8, 10, 5, 9, 12

Empty tapes:

Tape 3:
Tape 4:
Tape algorithm: toy example - merge

Tape 1: 1, 4, 6, 2, 7, 11
Tape 2: 3, 8, 10, 5, 9, 12

Merge into:
Tape 3: 1, 3, 4, 6, 8, 10
Tape 4: 2, 5, 7, 9, 11, 12
Tape algorithm: toy example – rewind 4 tapes

Tape 1: $1, 4, 6, 2, 7, 11$
Tape 2: $3, 8, 10, 5, 9, 12$

Merge into:

Tape 3: $1, 3, 4, 6, 8, 10$
Tape 4: $2, 5, 7, 9, 11, 12$
Tape algorithm: toy example - final merge

Input:

Tape 3: \(1, 3, 4, 6, 8, 10\)

Tape 4: \(2, 5, 7, 9, 11, 12\)

Merge:

Tape 1: \(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\)
Tape algorithm: while rewinding the tape

- Sorted runs:
  1, 4, 6, 3, 8, 10, 2, 7, 11, 5, 9, 12

Tape 1: 1, 4, 6, 2, 7, 11
Tape 2: 3, 8, 10, 5, 9, 12

While rewinding Tape 1 and Tape 2, merge backwards—and write on Tapes 3, 4 in descending order:

Tape 3:

Tape 4:
Tape algorithm: while rewinding the tape

• Sorted runs:
  \[1, 4, 6, 3, 8, 10, 2, 7, 11, 5, 9, 12\]

While rewinding Tape 1 and Tape 2, merge backwards—and write on Tapes 3, 4 in descending order:

Tape 3: \[12, 11, 9, 7, 5, 2\]
Tape 4: \[10, 8, 6, 4, 3, 1\]
Tape algorithm: while rewinding the tape

Inputs tapes:

- **Tape 3:** 12, 11, 9, 7, 5, 2
- **Tape 4:** 10, 8, 6, 4, 3, 1

Merge:

Tape 1:
Tape algorithm: while rewinding the tape

Inputs tapes:

Tape 3: 12, 11, 9, 7, 5, 2
Tape 4: 10, 8, 6, 4, 3, 1

Merge:
Tape 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Unfortunately, we cannot read from the hard disk backwards!
Summary

- Disk constraints require different algorithms which reduce the number of disk I/Os.
- The fastest and widely used sorting algorithm for large inputs is 2PMMS.
Ass 3 and exam questions

• How to implement efficient random shuffling of very large inputs?
• How to incorporate duplicate elimination into the sorting algorithm?
• What data structure to use best for the selection of the smallest key under current pointer in all input buffers?