CMPS340 File Processing
Estimating the running time for the
merging phase of external (disk) sorting

The first phase of the classic external *Polyphase Merge* sorting algorithm results in there being a collection of *initial sorted runs* (or *segments*) on the disk, each having size (approximately) equal to the size of available RAM.[1] Each sorted run is a (sorted) portion of the file that is to be sorted.

The second phase of the algorithm involves merging the sorted runs together in order to produce —after one or more *passes*— a single sorted run. The number of passes required depends upon how many sorted runs are merged together "at a time".

For example, if we begin with 80 sorted runs and merge 80 runs at a time, one merging pass suffices. On the other hand, if we merge 10 runs at a time during the first pass, the result would be eight sorted runs. (The first ten runs are merged into one sorted run, then the next ten runs are merged into a second sorted run, etc., etc.) On the second pass, we could merge all eight runs together at once. Alternatively, we could merge them two at a time, which would result in four runs, necessitating at least one more merging pass. The arithmetic doesn't come out perfectly, but we could also merge the 80 runs 9 at a time (with the last group consisting of only 8 runs), yielding 9 runs, which could then be merged in one or more subsequent passes.

Let us assume that, as we merge a group of sorted runs, we write the resulting sorted run to a second disk drive; this means that input and output operations can be overlapped, for the most part, making it reasonable for us to count only the time taken by input operations. (If we have only one disk, we can estimate the running time for input and then double it to arrive at an estimate of the total time needed.) Based on the size of the file and the speed at which the disk reads/writes data, we can compute t, the time necessary to transfer all the data in the file to (from, respectively) RAM from (to, respectively) disk (not including time required by seeks). Specifically, if the file occupies F blocks and the block transfer time is btt, then $t = F \cdot btt$. We also know $s + r$, the "average" time required for seeking plus rotational delay.

The initial sorted runs are merged together in $n > 0$ passes; for i satisfying $0 < i \le n$ we use m_i to denote the number of sorted runs that exist as the i-th pass begins and S_i to denote the (common) size of those sorted runs. F and R denote the size of the original file and of available RAM, respectively. (We can measure these in terms of bytes or blocks, for example. Here we use the latter.) The sum total of the sizes of the sorted runs should equal the size of the file; thus, we have, for i satisfying $0 < i \le n$, $S_i \cdot m_i = F$. Equivalently,

$$m_i = \frac{F}{S_i} \tag{1}$$

---

[1]Note that there is a technique called *Replacement Selection* that can be used when (at least) two disk drives are available —one for input and another for output— in order to produce initial sorted runs that are, on average, twice the size of available RAM. See Folk & Zoellick (or Volume 3 of Knuth's *Art of Computer Programming*) for details.

The running time for the i-th merging pass is given by

$$t_i = k_i \cdot m_i \cdot (s + r) + t_f \tag{2}$$

where k_i is the number of seeks performed on each sorted run during pass i. *Remark:* The second term on the right-hand side of (2), t, accounts for the time required to transfer all the data in the file from disk to RAM. (During a merging pass, every record in every sorted run is read into RAM.) The first term accounts for the time required by all the seeking that occurs during the pass. The number of seeks performed during the i-th pass is clearly equal to the product of the number of runs it started with, m_i, and the number of seeks made to each run, k_i. *End of remark*

Now, the value of k_i depends upon S_i, R, and P_i, where P_i is the number of runs merged at a time during pass i. (We use the term "P-way merging" to refer to merging P runs at a time.) In carrying out P-way merging, we use a buffer of size R/P for each of the P runs involved. (Thus, the buffers occupy all of available RAM.[2]) Initially, each buffer is filled with the first chunk of data (of size R/P) from its corresponding run. During the merging process, the data in the buffers is "consumed"; whenever a buffer becomes empty, it is re-filled by obtaining the next chunk (of size R/P) from the corresponding run, which requires a seek. Since each run has size S_i and is brought into RAM in chunks of size R/P_i (the buffer size), the number of chunks brought into RAM from each run (and hence the number of seeks made to each run) must be $\frac{S_i}{R/P_i}$. Simplifying, we get

$$k_i = \frac{S_i \cdot P_i}{R} \tag{3}$$

Making use of (1) and (3), we get that

$$
\begin{aligned}
k_i \cdot m_i &= \frac{S_i \cdot P_i}{R} \cdot \frac{F}{S_i} \\
&= P_i \cdot \frac{F}{R}
\end{aligned}
$$

Substituting for $k_i \cdot m_i$ in (2), we get

$$t_i = P_i \cdot \frac{F}{R} \cdot (s + r) + t_f$$

Now, the total running time T is the sum of the running times of the n merging passes; that is, $T = \sum_{1 \le i \le n} t_i$. Thus, we get

$$
\begin{aligned}
T &= \sum_{1 \le i \le n} t_i \\
&= \sum_{1 \le i \le n} (P_i \cdot \frac{F}{R} \cdot (s + r) + t_f) \\
&= \sum_{1 \le i \le n} (P_i \cdot \frac{F}{R} \cdot (s + r)) + \sum_{1 \le i \le n} t_f \\
&= \frac{F}{R} \cdot (s + r) \cdot \sum_{1 \le i \le n} P_i \quad + \quad n \cdot t_f
\end{aligned}
$$

---

[2]Some RAM must also be used to provide, say, two output buffers (for double-buffering), but, as these buffers can be small, we will ignore this.

Obviously, our choices for the P_i's determine how many passes are necessary to complete the job. The number of sorted runs remaining after the 1st pass will be $\frac{m_1}{P_1}$, the number remaining after the 2nd pass will be $\frac{m_1}{P_1 \cdot P_2}$, and, generally, after the i-th pass, $\frac{m_1}{\hat{P}_i}$ (where $\hat{P}_i = \prod_{1 \leq j \leq i} P_j$). Thus, the product $\prod_{1 \leq i \leq n} P_i$ equals (or exceeds) m_1.

Suppose that, for a fixed number of passes n, we want to know the best choices for P_1, ..., P_n that will complete the job of merging the initial sorted runs. For example, suppose that $n = 2$ and $m_1 = 72$. Should we choose $P_1 = 12$ and $P_2 = 6$, or would $P_1 = 9$ and $P_2 = 8$ be better? How about $P_1 = 4$ and $P_2 = 18$? Letting $a = \frac{F}{R} \cdot (s + r)$ and $b = n \cdot t_f$, we have

$$T = a \cdot \sum_{1 \leq i \leq n} P_i \;\; + \;\; b$$

Clearly, then, to minimize T we want to choose the P_i's to be such that their sum is as small as possible. For the choices just mentioned, the best is $P_1 = 9$ and $P_2 = 8$ because $9 + 8 = 17$, as compared to $12 + 6 = 18$ and, worse still, $4 + 18 = 22$. This illustrates a more general point, which is that the best choice for the P_i's is such that they are as close in value to one another as possible. Indeed, if we could pick real numbers rather than integers, we would make $P_i = \sqrt[n]{m_1}$ for all i.

To a first approximation, then, we may characterize $T(n)$, the estimated running time of merging a set of sorted runs using n passes, as follows:

$$
\begin{aligned}
T(n) \;\; &= \;\; \frac{F}{R} \cdot (s + r) \cdot n \cdot \sqrt[n]{m_1} \;\; + \;\; n \cdot t_f \\[2mm]
&= \;\; n\left(t_f \;\; + \;\; \frac{F}{R} \cdot (s + r) \cdot \sqrt[n]{m_1}\right)
\end{aligned}
$$

Assuming, as in the usual case, $m_1 = F/R$, we get

$$
\begin{aligned}
T(n) \;\; &= \;\; n\left(t_f \;\; + \;\; \frac{F}{R} \cdot (s + r) \cdot \sqrt[n]{\frac{F}{R}}\right) \\[2mm]
&= \;\; n\left(t_f \;\; + \;\; (s + r) \cdot \left(\frac{F}{R}\right)^{1 + \frac{1}{n}}\right)
\end{aligned}
$$

It turns out that, given the values of s, r, R, etc., as determined by the computer hardware of today, the best choice for n (number of passes) is almost always small (say $n \leq 3$). Thus, by evaluating T for a few small values of n (say $n = 1, 2, 3, 4$), we can determine the optimum choice for n, which determines (more or less) the best choices for the P_i's.