# Efficient Optimal Pagination of Scrolls

*L. L. Larmore*[†] and *D. S. Hirschberg*[‡]

Technical Report #224

April, 1984

**Abstract.** Diehr and Faaland developed an algorithm that finds the minimum sum of key length pagination of a scroll of $n$ items, and which uses $O(n \lg n)$ time and $O(n)$ space, solving a problem posed by McCreight. An improved algorithm is given which uses $O(n)$ time and $O(n)$ space.

---

This research was supported in part by a California State University PAID grant and National Science Foundation Grant MCS-82-00362.

[†]Department of Computer Science, California State University, Dominguez Hills, CA 94707.
[‡]Department of Information and Computer Science, University of California, Irvine, CA 92717.

# Efficient Optimal Pagination of Scrolls

*L. L. Larmore*[†]
California State University, Dominguez Hills
*D. S. Hirschberg*[‡]
University of California, Irvine

## Abstract

Diehr and Faaland developed an algorithm that finds the minimum sum of key length pagination of a scroll of $n$ items, and which uses $O(n \lg n)$ time and $O(n)$ space, solving a problem posed by McCreight. An improved algorithm is given which uses $O(n)$ time and $O(n)$ space.

## Introduction

Suppose that we are given a scroll of $n$ items of varying length. Let $w_i > 0$ be the length of the $i$th item. A *boundary sequence* is a sequence $0 = s_0 < s_1 < \ldots < s_{v+1} = n+1$ such that $p_{min} \leq \Sigma_{s_{k-1}<i<s_k} w_i \leq p_{max}$ for all $1 \leq k \leq v+1$, where $0 \leq p_{min} < p_{max}$ are fixed. The *length* of that boundary sequence is defined to be $\Sigma_{1 \leq k \leq v} w_{s_k}$. McCreight [2] asks whether we can "quickly" find a boundary sequence of minimum length.

Diehr and Faaland [1] develop an algorithm which finds the minimum length boundary sequence in $O(n \lg n)$ time, using $O(n)$ storage. In this paper, we introduce an algorithm which requires both $O(n)$ time and $O(n)$ space.

For convenience, assign any positive value, say 1, to $w_{n+1}$ and $w_0$.

Define $Gap(a,b)$ as the sum of the lengths of the scroll items, $w_i$, *strictly between*

This research was supported in part by a California State University PAID grant and National Science Foundation Grant MCS-82-00362.
[†]Department of Computer Science, California State University, Dominguez Hills, CA 94707.
[‡]Department of Information and Computer Science, University of California, Irvine, CA 92717.

the $a^{th}$ and the $b^{th}$ items. Note that $Gap(a,a+1) = 0$. Define $Gap(a,a) = -w_a$.

Define boolean function $Page(a,b)$ to be *true* iff $p_{min} \leq Gap(a,b) \leq p_{max}$.

For any $0 \leq a \leq b \leq n+1$, we define an ***admissable path*** from $a$ to $b$ to be a sequence $s_0, s_1, \ldots s_v$ such that $Page(s_{k-1},s_k)$ for each $0 < k \leq v$. The *length* of that path is $\Sigma_{1 \leq k \leq v} w_{s_k}$. If there exists an admissable path from 0 to $j$, we say that $j$ is *accessable.*

For any $0 \leq i \leq n+1$, define $f(i)$ to be the minimum length of all paths from 0 to $i$. If $i$ is inaccessable, let $f(i) = \infty$.

For each $0 < i \leq n+1$ such that $Page(k,i)$ for some $k$, define $\rho(i)$ to be the unique number which satisfies the following three conditions:

(i) $Page(\rho(i),i)$

(ii) $f(\rho(i))$ is minimized subject to (i)

(iii) $\rho(i)$ is minimized subject to (i) and (ii)

If there is no $k$ for which $Page(k,i)$ is *true*, then $\rho(i)$ is undefined. Also, $\rho(0)$ is undefined.

Computation of $f$ and $\rho$ clearly suffices to find the minimum length boundary sequence. A boundary sequence exists if and only if $f(n+1) < \infty$, and the minimum length boundary sequence can be found (in reverse order) by using $\rho$.

## The Algorithm

### Main Procedure

Step 1. Initialization:
Compute $Sum[i] = \Sigma_{k \leq i} w[k]$, $0 \leq i \leq n+1$
$next[i] \leftarrow -1$, $0 \leq i \leq n+1$
$f[0]$, $rho \leftarrow 0$

Step 2. For $i := 1$ to $n+1$ do Steps 3 through 6

Step 3. Advance_rho

Step 4. If *not Page*(*rho*,*i*) then $f[i] \leftarrow \infty$

Step 5. Otherwise

$f[i] \leftarrow f[rho] + w[i]$

$\rho[i] \leftarrow rho$

Step 6. Update_arrays

Step 7. Halt

**Procedure Advance_rho**

Step 1. While $Gap(rho,i) > p_{max}$ do

$rho \leftarrow rho+1$

Step 2. While $rho < next[rho]$ and $Gap(next[rho],i) \geq p_{min}$ do

$rho \leftarrow next[rho]$

Step 3. Return

**Procedure Update_arrays**

Step 1. $j \leftarrow i-1$

Step 2. While $f[j] > f[i]$ do

$next[j] \leftarrow i$

$j \leftarrow backup[j]$

Step 3. $backup[i] \leftarrow j$

Step 4. Return

## Proof of Correctness

It is important to distinguish between the functions $f(i)$ and $\rho(i)$ on the one hand, which are defined abstractly, and the arrays $f[i]$ and $\rho[i]$, whose values are assigned dynamically during execution of the algorithm.

We remind the reader that, for all $0 < i \leq n+1$, either $f(i) = \infty$ or $f(i) = f(\rho(i)) + w_i$.

Intuitively, the algorithm works as follows. *rho* is a running "temporary" $\rho(i)$, which never decreases. When *rho* is too small because $Gap(rho,i) > p_{max}$, *rho* is incremented by 1 until *Gap* is small enough. We then need to increase *rho*, minimizing the $f$ value, thus obtaining $\rho(i)$. In [1], a heap of possible values is maintained, and it

takes $\Theta(\lg n)$ time to find $\rho(i)$. In our algorithm, the pointer *next* tells us where to look next. Even though it might take $\Theta(n)$ time to find $\rho(i)$ for a particular $i$, the total time for these searches over all $i$ is still only $O(n)$, since *rho* never decreases. The pointer array *backup* is used for updating *next*, and also for updating itself.

Our method of proof is to define a loop invariant, and to prove inductively that the loop invariant holds after any number of iterations of the loop of Main.

*Loop invariant.* For any $0 \leq i \leq n+1$, the following conditions hold after $i$ iterations of the loop of Main:

L1($i$): If $\rho(i)$ is defined, *rho* $= \rho(i)$. Otherwise, *rho* is the smallest $j$ such that $Gap(j,i) \leq p_{max}$.

L2($i$): For all $0 \leq j \leq i$, $f[j] = f(j)$.

L3($i$): For all $0 \leq j \leq i$, if $\rho(j)$ is defined, $\rho[j] = \rho(j)$. Otherwise, $\rho[j]$ is undefined.

L4($i$): For all $0 \leq j \leq i$, *next*$[j]$ is the smallest $j < k \leq i$ such that $f(k) < f(j)$, provided there is such a $k$. Otherwise, *next*$[j] = -1$.

L5($i$): For all $0 < j \leq i$, *backup*$[j]$ is the largest $0 \leq k < j$ such that $f(k) \leq f(j)$.

It is clear that the loop invariant holds initially, i.e., after execution of Step 1 of Main, i.e., when $i = 0$. Assume, now, that the loop invariant holds after $(i-1)$ iterations of the loop, $i \geq 1$. We show it still holds after one more iteration.

Define integers $0 \leq \alpha_i \leq \beta_i \leq i$ as follows. $\alpha_i$ is as small as possible such that $Gap(\alpha_i,i) \leq p_{max}$, and $\beta_i$ is as small as possible such that $Gap(\beta_i,i) < p_{min}$. Note that $\{\alpha_i\}$ and $\{\beta_i\}$ are monotone increasing sequences. It is seen that $\rho(i)$ is defined if and only if $\alpha_i < \beta_i$, and that, if $\rho(i)$ is defined, the following conditions hold:

(i) $\alpha_i \leq \rho(i) < \beta_i$

(ii) $f(\rho(i))$ is minimum subject to (i)

(iii) $\rho(i)$ is minimum subject to (i) and (ii)

*Proof of L1(i).* If $\rho(i-1)$ is not defined, *rho* $= \alpha_{i-1}$ before execution of Step 3 of

Main. After completion of Step 1 of Advance_rho, $rho = \alpha_i$. If $\rho(i-1)$ is defined, $rho = \rho(i-1)$ before execution of Step 3 of Main. After completion of Step 1 of Advance_rho, $rho$ equals $\rho(i-1)$ or $\alpha_i$, whichever is larger.

Consider two cases, $\rho(i)$ is undefined and $\rho(i)$ is defined.

If $\rho(i)$ is undefined, then $rho = \alpha_i$ after completion of Step 1 of Advance_rho and $\alpha_i = \beta_i$. Step 2 will not loop at all since $Gap(rho,i) < p_{min}$. Therefore, for any $k < rho = \alpha_i$, $Gap(k,i) > p_{max}$. Thus L1($i$) is satisfied.

On the other hand, suppose $\rho(i)$ is defined. We define a loop invariant on the iterations of Step 2 of Advance_rho:

AL1: $rho < \beta_i$

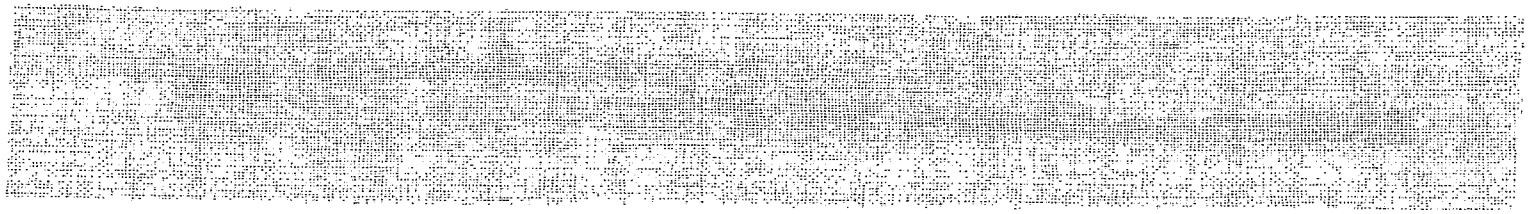
AL2: $f(k) > f(rho)$ for all $\alpha_i \leq k < rho$

If $rho = \alpha_i$ after Step 1, then $rho = \alpha_i < \beta_i$ so AL1 is satisfied and AL2 is vacuously satisfied. On the other hand, if $rho = \rho(i-1)$ after Step 1, AL1 is satisfied since $rho < \beta_{i-1} \leq \beta_i$, while AL2 is satisfied by definition of $\rho(i-1)$, since $\alpha_{i-1} \leq \alpha_i$. We now show that AL is maintained by each iteration of Step 2 of Advance_rho.

We note that *Gap* is monotone decreasing in its first parameter. Suppose $rho < next[rho]$ and $Gap(next[rho],i) \geq p_{min}$. Then $next[rho] < \beta_i$ by definition of $\beta_i$ and the monotonicity of *Gap*. Also, $f(next[rho]) < f(rho) \leq f(k)$ for all $rho < k < next[rho]$, by L4($i$-1). And, $f(next[rho]) < f(rho) < f(k)$ for all $\alpha_i \leq k < rho$, by the previous AL2. Thus the assignment in Step 2 of Advance_rho maintains the loop invariant AL.

Step 2 of Advance_rho will complete only when either $next[rho] = -1$, which means that $f(k) \geq f(rho)$ for all $rho < k < i$, or $next[rho] \geq \beta_i$. In either case, $f(k) \geq f(rho)$ for all $rho < k < \beta_i$. Together with AL2, this shows that $f(rho)$ is minimum in the range $[\alpha_i, \beta_i - 1]$ and so $rho = \rho(i)$. Therefore L1($i$) is satisfied.

*Proof of L2(i).* By L2($i$-1), $f[j] = f(j)$ for all $j < i$. We note that $i$ is accessable if



and only if *Page(rho,i)* and *rho* is accessable. If $i$ is accessable, $f(i) = f(rho)+w_i$. $f[i]$ is set to that value in Step 5 of Main.

*Proof of L3(i).* In the proof of L1($i$), we established that $rho = \rho(i)$ if and only if $\rho(i)$ is defined, i.e., if and only if *Page(rho,i)*. $\rho(i)$ is set to *rho* in Step 5 of Main.

*Proof of L4(i) and L5(i).* We define a loop invariant for Step 2 of Update_arrays:

LU1: $next[j] = -1$ (the sentinel value).

LU2: For all $j < k < i$, $f[k] > f[i]$.

LU3: For all $j < k < i$, $next[k]$ has its correct final value.

After execution of Step 1 of Update_arrays, $j = i$–1. By L4($i$–1), $next[i\text{–}1] = -1$. Thus LU1 holds. LU2 and LU3 hold vacuously.

Suppose LU holds before an iteration of Step 2. We show that it still holds after that iteration.

Since the loop is iterating, we have that $f[i] < f[j]$. Also, $f[k] \geq f[j]$ for all $j < k < i$, by LU1 which states that $next[j] = -1$, and by L4($i$–1). Thus, the correct final value of $next[j]$ should be $i$, by definition of *next*. Step 2 makes the correct assignment. It is already true that $next[k]$ has the correct final value for all $k$ where $backup[j] < k < j$, by L4($i$–1) and L5($i$–1), since $f[k] > f[j]$ in that range. By previous LU3, $next[k]$ is already the correct final value, for all $j < k < i$. Thus, $next[k]$ will be the correct final value for all $k$ in the range $backup[j] < k < i$. Thus, after the assignment $j \leftarrow backup[j]$, LU3 is preserved.

Since the loop is iterating, $f[j] > f[i]$. By LU2, $f[k] > f[i]$ for all $j < k < i$. By L5($i$–1), $f[k] > f[j]$ for $backup[j] < k < j$. Therefore, $f[k] > f[i]$ for $backup[j] < k < i$. The assignment $j \leftarrow backup[j]$ thus preserves LU2.

After the first assignment of Step 2, $next[j] = i$ and this is the correct assignment as shown two paragraphs above. As a result, $f[k] \geq f[j]$ for all $j < k < i$. By L5($i$–1),

$f[k] > f[j]$ for $backup[j] < k < j$, and $f[backup[j]] \leq f[j]$. Therefore, for $backup[j] < k \leq j$, $f[k] \geq f[j] \geq f[backup[j]]$. Combining this last inequality with the first inequality of this paragraph, $f[k] \geq f[backup[j]]$ for $backup[j] < k < i$. Therefore, by L4($i$–1), $next[backup[j]] = -1$. Thus, LU1 is preserved when $j$ is reassigned.

We have therefore shown that LU is invariant.

When Step 3 of Update_arrays is executed, $f[j] \leq f[i]$ since Step 2 no longer is iterating, and also $f[k] > f[i]$ for all $j < k < i$ by LU2. Thus Step 3 asssigns the correct value of $backup[i]$. By L5($i$–1), all previous values of *backup* are correct, and therefore L5($i$) is true.

We are left only with verification of L4($i$). $next[i] = -1$ since it was never reset and that is its correct value. For all $k < backup[i]$, L4($i$–1) assures that $next[k]$ is correct, since the fact that $f[i] \geq f[backup[i]]$ rules out $i$ as a possible value for $next[k]$, and there is no other new candidate. For $backup[i] < k < i$, $next[k]$ is correct by LU3 and the fact that the final value of $j$ in Update_arrays is $backup[i]$. It only remains to show that $next[backup[i]]$ has its correct value; By LU1, we know it is still –1.

By L4($i$–1), for all $backup[i] < k \leq i$–1, $f[k] \geq f[backup[i]]$. The only possible remaining candidate for $next[backup[i]]$ is thus $i$, which is ruled out since $f[i] \geq f[backup[i]]$. Therefore $next[backup[i]] = -1$ is correct. We conclude that L4($i$) holds.

Finally, the algorithm is correct by L2($n$+1) and L3($n$+1).

## Proof of Linear Time and Space Complexity

*Storage.* Only five arrays are needed: *Sum*, $f$, $\rho$, *next*, and *backup*. Each of these is linear. The values of *Gap* and *Page* can be computed as needed in $O(1)$ time each, using *Sum*.

*Time for the Main Algorithm.* Step 1 takes $O(n)$ time. The main loop (Steps 3 through 6) is executed $n+1$ times. We look at each step from 3 to 6 separately.

Step 3 is executed $n$ times, and each execution is in $O(n)$ time. But we show (below) that the total time of all those executions is still $O(n)$.

Steps 4 and 5 are clearly done in $O(1)$ time, for a total of $O(n)$ time.

Step 6 is executed $n$ times, and each execution is in $O(n)$ time. But we show (below) that the total time of all those executions is still $O(n)$.

*Time for procedure Advance_rho.* This procedure is called $n+1$ times. Each iteration of Step 1 or Step 2 increases the value of *rho*, which is bounded above by $n+1$. *rho* never is decreased. Therefore, the total number of iterations of Step 1 and Step 2 together, over all calls of the procedure, cannot exceed $n$. Thus the total execution time for procedure Advance_rho summed over all calls is $O(n)$.

*Time for procedure Update_arrays.* This procedure is called $n+1$ times. Thus, Steps 1 and 3 are executed a total of $n+1$ times each. Each time Step 2 iterates, the value of some *next*[$j$] is changed from being $-1$ (the sentinel) to a value more than $j$. Since the values of *next* are never reassigned, it is clear that the total number of times Step 2 iterates, over all calls, cannot exceed $n+1$. It follows that the total execution time for procedure Update_arrays summed over all calls is $O(n)$.

## References


[1] Diehr, G. and Faaland, B. Optimal pagination of B-trees with variable-length items. *Comm. ACM 27*, 3 (March 1984), 241-247.

[2] McCreight, E.M. Pagination of B*-trees with variable-length records. *Comm. ACM 20*, 9 (Sept. 1977), 670-674.