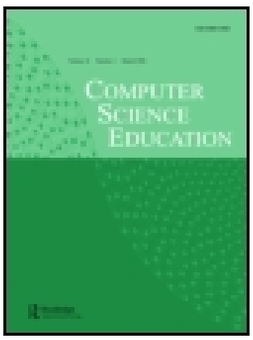

Computer Science Education

ISSN: 0899-3408 (Print) 1744-5175 (Online) Journal homepage: https://www.tandfonline.com/loi/ncse20

Planning readings: a comparative exploration of basic algorithms

Justus H. Piater

To cite this article: Justus H. Piater (2009) Planning readings: a comparative exploration of basic algorithms, Computer Science Education, 19:3, 179-192, DOI: 10.1080/08993400903255226

To link to this article: https://doi.org/10.1080/08993400903255226

Published online: 14 Oct 2009.

Computer Science Education
Vol. 19, No. 3, September 2009, 179–192

Routledge
Taylor & Francis Group

Planning readings: a comparative exploration of basic algorithms

Justus H. Piater*

Department of Electrical Engineering and Computer Science, Université de Liège, Liège, Belgium

(*Received 22 December 2008; final version received 31 July 2009*)

Conventional introduction to computer science presents individual algorithmic paradigms in the context of specific, prototypical problems. To complement this algorithm-centric instruction, this study additionally advocates problem-centric instruction. I present an original problem drawn from students' life that is simply stated but provides rich discussions of different approaches. It lends itself to a wide range of didactic means from individual or group study to whole class discussions under various levels of guidance by the instructor. I suggest diverse algorithms for solving it, covering some of the most important algorithmic paradigms. Some of these algorithms (greedy, divide-and-conquer) do not produce optimal solutions but may nevertheless have their merits in practice. The best algorithms are illustrative instances of some of the most sophisticated paradigms introduced in undergraduate curricula (dynamic programming, graphs).

Keywords: algorithms; greedy method; divide-and-conquer; dynamic programming; graph-based methods

1. Introduction

Algorithms lie at the heart of computer science. A central part of undergraduate computer science curricula is an introduction to the most important types of algorithms, such as the greedy method, divide-and-conquer, dynamic programming, and graph-based algorithms. Such distinct basic algorithmic techniques differ in the classes of problems they address. Therefore, most popular algorithm textbooks (Cormen, Leiserson, Rivest, & Stein, 2001; Goodrich & Tamassia, 2005; Sedgewick, 2003) introduce them separately using example problems designed to represent typical instances for each of these techniques. This approach has the advantage of presenting algorithms in the context of typical use cases, allowing them to play to their full strengths and highlighting their distinctive properties.

Although such joint introduction to individual problems and their algorithmic solutions is an important pedagogical technique, it does not represent a real-life situation. In reality, problems present themselves without their associated algorithms, and it is the task of the computer scientist to identify suitable algorithmic approaches. However, transfer of specific knowledge to related tasks

*Email: justus.piater@ulg.ac.be

ISSN 0899-3408 print/ISSN 1744-5175 online

DOI: 10.1080/08993400903255226
http://www.informaworld.com

is not automatic but requires additional training or practice (Broudy, 1977; Ericsson, Chase, & Faloon, 1980).

In my experience, for many students, mastery of individual algorithms is insufficient to solve novel problems. For example, I observe that some distinctions between algorithms appear quite subtle to novices: Both greedy and dynamic programming algorithms make sequential choices; both divide-and-conquer and dynamic programming break a problem into smaller subproblems. If distinct types of algorithms are introduced in isolation and on distinct problems, students are required to simultaneously grasp both the characteristic distinctions among the algorithms and those among the problems, while making causal links between them and without getting distracted by the plentiful but unimportant distinctions among the illustrating problems. These difficulties may obstruct the generalized features that characterize the correspondences between algorithms and problems, even if the algorithms are well understood.

I therefore argue that the conventional way of introducing algorithms should be complemented by comparative studies of algorithms on shared problems (Bransford, Franks, Vye, & Sherwood, 1989; Schwartz, Lin, Brophy, & Bransford, 1999). By studying to what extent different algorithms apply to one given problem and comparing their behavior, students may more readily discern fundamental distinctions between the types of algorithms without the added difficulty of abstracting away from different example problems.

Moreover, the study of different solutions to a single problem emulates a real-life problem solving process that engages students interactively: Given a problem, students explore various ways of solving it, and produce well-motivated solutions by themselves. These activities are known to be effective for learning, but play only a subordinate role in most conventional teaching of algorithms.

Such a study requires example problems that are sufficiently rich to admit various algorithmic approaches that highlight their distinctive properties, are sufficiently well structured to admit elegant solutions in terms of at least one classical type of algorithm, and are sufficiently simple to describe such that learning focuses on the algorithms rather than the problem. Such multi-algorithmic problems are substantially harder to design than problems that illustrate a single algorithmic approach.

In the following section, I present a didactic problem that is drawn from daily student life, simple to describe and yet nontrivial to solve. In my second-year, university-level teaching of data structures and algorithms, I challenge students to develop and explore various types of solutions *via* homework assignments and classroom discussions. In subsequent sections, I discuss some suboptimal and optimal solutions such as those students may produce or may be guided towards. They include brute-force, greedy, divide-and-conquer, dynamic programming and graph-based algorithms.

As is often the case in practice, the brute-force approach is intractable, and the relatively simple greedy and divide-and-conquer algorithms produce suboptimal solutions. The graph-based method is interesting in that it draws on several sophisticated data structures and algorithms, putting theoretical course content into an illustrative application context. The method of choice will be a dynamic programming algorithm that achieves the same asymptotic running time as the graph-based algorithm, but gives rise to an extremely simple and elegant implementation. This outcome is particularly desirable since, for novices, dynamic

programming is typically the hardest concept to grasp among the algorithmic paradigms traditionally covered in undergraduate curricula.

2. The problem: planning readings

In preparing for an exam, a student is faced with a voluminous textbook containing N chapters that she needs to cover in P days. Each chapter n constitutes a workload of w_n. The chapters are best covered in order, and no chapters are to be split across different days. The objective is to find a schedule that assigns chapters to days such that the student's workload is distributed as evenly as possible over the P days. Of course, this problem is only interesting if $N > P$.

Formally, we are given a sequence S of N elements of weights $w_n \geq 0$, $n = 1, \ldots, N$. We seek a partitioning

$$0 = n_0 \leq n_1 \leq \cdots \leq n_P = N \tag{1}$$

of S into P partitions that minimizes a cost function C, where n_p is the last chapter to be read on day p.

One attractive way to specify a cost function C that formalizes the notion of workload balancing is to aim for the average workload every day, and to minimize the squared deviation from this daily milestone, that is,

$$C_{\text{avg}}^2 = \sum_{p=1}^{P} \left(\bar{w} - \sum_{n=n_{p-1}+1}^{n_p} w_n \right)^2 \tag{2}$$

where $\bar{w} = \frac{1}{P} \sum_{n=1}^{N} w_n$.

The squared (as opposed to, say, the absolute) deviation has the desirable property of penalizing large deviations more heavily than small deviations. A typical student will try to avoid huge readings.

Another appealing aspect of this squared cost function is that it is actually independent of the constant $\bar{w}$. To show this, let us simplify notation by calling the (linear) cost of day p

$$C_p = \sum_{n=n_{p-1}+1}^{n_p} w_n, \tag{3}$$

rewrite the average squared cost (2) as

$$C_{\text{avg}}^2 = \sum_{p=1}^{P} (\bar{w} - C_p)^2 = P\bar{w}^2 - 2\bar{w} \sum_{p=1}^{P} C_p + \sum_{p=1}^{P} C_p^2$$

and note that $\sum_{p=1}^{P} C_p = \sum_{n=1}^{N} w_n$ does not depend on the partitioning. Thus, under our squared cost function, balancing the workload over the study period turns out to be equivalent to minimizing the daily workload (by replacing $\bar{w}$ by 0), which is certainly not unattractive to students!

This gives rise to the equivalent but simpler cost function

$$C^2 = \sum_{p=1}^{P} C_p^2, \tag{4}$$

where C_p^2 is the (squared) cost of day p.

3. Brute force

Conceptually, the easiest way to find an optimal partitioning is to enumerate all reasonable partitionings and choose the cheapest among them.

If $N \gg P$, the number $s(N,P)$ of distinct partitionings is easily found to be at least exponential in P, since for each choice of n_i there are multiple independent choices for each of the remaining $n_{j\neq i}$.

For a more careful count, observe that if $P = 1$ or $P \geq N$ there is only one distinct solution; all schedules that assign exactly zero or one chapter to each day are considered equivalent. Otherwise, let us read n chapters on the first day, and enumerate the partitionings for the $N-n$ remaining chapters over the remaining $P-1$ days. This algorithm for exhaustive enumeration of the possible partitionings gives rise to the recurrence relation

$$\begin{aligned} s(N,P) &= \begin{cases} 1 & \text{if } N \leq P \text{ or } P \leq 1, \\ \sum_{n=1}^{N-P+1} s(N-n, P-1) = \sum_{n=P-1}^{N-1} s(n, P-1) & \text{otherwise,} \end{cases} \\ &= \begin{cases} 1 & \text{if } P \leq 1, \\ 1 + \sum_{n=P}^{N-1} s(n, P-1) & \text{otherwise.} \end{cases} \end{aligned} \tag{5}$$

Another way of counting the number of partitionings, albeit without an algorithm for enumerating them, is to ask how many different ways there are to choose a set of separators $\{n_P\}_{p=1}^{P-1}$ from the set of $N-1$ possible chapters (1). This number is clearly the binomial coefficient

$$s(N,P) = \binom{N-1}{P-1} \tag{6}$$

which can be shown to be equal to the recurrence relation (5), see Lemma 1.1 in the Appendix. The proof by induction is straightforward with a slightly interesting twist.

4. Two suboptimal greedy algorithms

Our original objective function (2) states that we seek to do the average amount of reading every day. This directly leads to a greedy algorithm that chooses, for each day $p = 1, \ldots, P-1$, the target chapter n_p that minimizes the daily cost

$$D_p^2 = (\bar{w} - C_p)^2. \tag{7}$$

However, does a sequence of greedy choices according to this local objective function minimize our global objective function (2), that is, $C^2_{\text{avg}} = \sum_{p=1}^{P} D_p^2$? A simple counterexample shows that this is not generally the case:

Example 4.1 Consider $P = 6$, $N = 8$ and all $w_n = 3$. Then, $\bar{w} = 4$, and our greedy approach will allocate one chapter to each day, leaving three chapters to the last day. The total cost of this solution is $C^2_{\text{avg}} = 5(4-3)^2 + (4-9)^2 = 30$. This is clearly suboptimal, since moving one of the last day's chapters to another day reduces the cost to $C^2_{\text{avg}} = 4(4-3)^2 + 2(4-6)^2 = 12$.

A straightforward way to implement this algorithm is, for each day p, to add one chapter at a time until C_p is minimized. Since all w_n are nonnegative, the optimum for day p is found by stopping as soon as C_p grows, and then backtracking one step. Since this procedure involves a linear traversal of the w_n, with each w_n visited at most twice, its running time is $O(N)$. Beyond the storage required for input and output, its additional space requirements are constant.

The obvious problem with the above algorithm is the myopic focus on the cost of the current day, which in Example 4.1 leads to an accumulation of underspent reading capacities until the last day. Can we not fix this by making daily greedy choices that minimize an amortized daily cost

$$E_p^2 = \left(p\bar{w} - \sum_{q=1}^{p} C_q \right)^2 \tag{8}$$

such that any accumulated imbalance can be leveled out over time?

Example 4.2 Using the data from Example 4.1, this would lead to the following daily readings:

day p	1	2	3	4	5	6
final daily chapter n_p	1	3	4	5	7	8
daily amortized cost ${E_p}^2$	1	1	0	1	1	0

Here, this procedure yields an even distribution of readings (with two chapters on days 2 and 5) that is easily seen to be optimal with a cost of $C^2_{\text{avg}} = 12$.

This improved algorithm can be implemented similarly to the above, and has the same asymptotic resource requirements. It is, alas, not generally optimal either, as the following counterexample shows.

Example 4.3 Reconsider Example 4.2, with the change of $w_8 = 9$, which entails $\bar{w} = 5$:

day p	1	2	3	4	5	6
final daily chapter n_p	2	3	5	7	8	8
daily amortized cost ${E_p}^2$	1	1	0	1	25	0

The cost of this solution – which assigns no reading to day 6 – is $C^2_{\text{avg}} = 3(5-6)^2 + (5-3)^2 + (5-9)^2 + 5^2 = 48$, while moving a chapter from a 2-chapter day to day 6 reduces the cost to $C^2_{\text{avg}} = 2(5-6)^2 + 3(5-3)^2 + (5-9)^2 = 30$.

In a similar vein, counterexamples can be found for any greedy algorithm of this type. In general, the problem is that the optimal solution for a given subrange of days may depend on the optimal solution for future subranges. In Example 4.3 it would have been better to initially make a locally suboptimal choice by saving a chapter for later such that all days could be assigned a reading. However, this is not something a greedy algorithm can do – all it does is make one choice at a time according to a local objective function. For a greedy algorithm to produce an optimal solution in terms of a global objective function, the local objective function must capture all relevant information such that a sequence of greedy choices with respect to it provably yields a globally optimal solution. As we have seen, it is not obvious how to specify such a local objective function for our reading planner.

5. A suboptimal divide-and-conquer algorithm

Instead of trying to solve our problem by a sequence of *local* choices, let us now look at the problem as a whole and try to break it down hierarchically. One obvious way to do this is to postulate that an optimal solution over the entire sequence of days can be composed of optimal solutions to the first and second half-sequences of days, where each such half-sequence is allocated half the weight of the chapters. A recursive application of this idea leads to the divide-and-conquer algorithm shown in Table 1, where on line 6, $C_{n,l} = \sum_{i=n}^{l} w_i$, and on line 7, rounding is done upward or downward according to the sign of the minimal difference found on line 6.

The calculation of n_{mid} on line 6 reflects the objective function (4), since, as we have seen there, balancing workloads amounts to minimizing squared workloads. Thus, line 6 is equivalent to

$$n_{\text{mid}} \leftarrow \arg\min_{l=n,\cdots,N} \left\{ C^2_{n;l} + C^2_{l+1,N} \right\}.$$

In practice, n_{mid} can be found in $O(log(N-n))$ time by binary search on an array of cumulative sums of the w_i ; this array needs to be constructed only once.

To analyze the running time of PLAN(1, N, 1, P), consider its binary recursion tree, which generally has $O(\log P)$ levels. At each level, a total of $O(P)$ separators is returned. Moreover, at level l, there are $O(2^l)$ binary searches within $O(\frac{N}{2^l})$ elements each (line 6), which total

$$\begin{aligned} O\left(\sum_{l=1}^{\log P} 2^l \log\frac{N}{2^l}\right) &= O\left(\sum_{l=1}^{\log P} \left(2^l \log N - l2^l\right)\right) \\ &\subseteq O\left(\log N \sum_{l=1}^{\log P} 2^l\right) \\ &= O(P \log N) \end{aligned} \tag{9}$$

operations for the entire algorithm. Thus, the total running time of PLAN(1, N, 1, P) is $O(N + P\log P + P\log N)$, which for $N > P$ is $O(N + P\log N)$. Even though this

Table 1. A (suboptimal) divide-and-conquer algorithm.

Input The first and last chapters n, N to assign to the days p through P.
Output A sequence of last chapters to read on consecutive days.

```
 1: function PLAN(n, N, p, P)
 2:     if N ≤ P then
 3:         return n, n + 1, ..., N[, N]_{l=N+1}^{P}
 4:     if P = 1 then
 5:         return N
 6:     n_mid ← arg min_{l=n,...,N} {|C_{n,l} − C_{l+1,N}|}
 7:     p_mid ← p + ROUND (P/2)
 8:     s_1 ← PLAN(n, n_mid, p, p_mid)
 9:     s_2 ← PLAN(n_mid + 1, N, p_mid + 1, P)
10:     return s_1,s_2
```

analysis is rather crude because it dropped the term $-l2^l$ in Equation (9), it still demonstrates the payoff of the binary search on the precomputed array of cumulative weights: Without it, linear searches of a total of N weights would have to be performed at each of the $\log P$ levels, leading to an asymptotic running time of $O(N \log P)$.

The space requirements are determined by the number of levels of recursion; at each level, a total of $O(P)$ separators is kept around. Thus, the overall space requirements are $O(N + P \log P)$.

Unsurprisingly, this divide-and-conquer algorithm does not generally yield optimal results, for reasons similar to the greedy algorithms discussed earlier.

Example 5.1 Consider $N = 5$, $w_n = 2$ for $n < 5$, $w_5 = 8$, and $P = 4$. Here, the algorithm in Table 1 will produce the sequence of separators 2, 4, 5, 5 with $C^2 = 2 \cdot 4^2 + 8^2 = 96$, whereas distributing the lightweight chapters over 3 days would reduce the cost to $C^2 = 4^2 + 2 \cdot 2^2 + 8^2 = 88$.

In an attempt to remedy this situation, one might try splitting the problem into equal numbers of chapters (as opposed to equal weights), but then again it is easy to give example configurations with grossly imbalanced weights that yield globally suboptimal solutions.

Concluding analogously to the preceding section, it is not obvious how to recursively subdivide the problem into independent subproblems in a way that a globally optimal solution can be efficiently constructed from their solutions.

6. A dynamic-programming algorithm

Our above experiences with the greedy and divide-and-conquer algorithms suggest that simple, local or hierarchical approaches will not lead to optimal solutions. Thus, let us reconsider the brute-force idea and ask whether there is a way to construct a globally optimal solution without enumerating all possible solutions. In dynamic programming, we specifically look for ways to construct a solution by successive augmentation of optimal solutions to subproblems. This reasoning immediately raises two questions:

(1) Does a globally optimal solution contain optimal solutions to subproblems?
(2) If so, how can we construct an optimal solution to a larger subproblem by augmenting a solution to a smaller subproblem?

One way to answer the first question is given by the following simple lemma.

Lemma 6.1 Optimal substructure. *In any optimal partitioning of $n_P = N$ elements into P partitions (1), the sub-partitioning of the first n_{P-1} elements into $P - 1$ partitions is optimal.*

Proof (By contradiction). Suppose we have an optimal partitioning of n_P elements into P partitions whose sub-partitioning of the first n_{P-1} elements into $P - 1$ partitions is not optimal. Then, we can improve the sub-partitioning without touching the last $n_P - n_{P-1}$ elements assigned to partition P. This reduces the cost (4) of the complete partitioning, which contradicts our supposition of optimality.

Given this optimal substructure, it is easy to answer our second question:

Corollary 6.2 Augmenting solutions to subproblems. *To find an optimal partitioning of N elements into P partitions, it suffices to enumerate all candidate partitionings composed of*

- *an optimal sub-partitioning of n elements into $P - 1$ partitions, and*
- *the remaining $N - n$ elements assigned to partition P,*

for $n = P - 1, \ldots, N - 1$, and to retain the minimum-cost candidate partitioning. Since, for $N \geq P$, any optimal solution will assign at least one element to each paritition, there are $N - P$ candidate partitionings.

Importantly, this corollary allows us to find a globally optimal partitioning without enumerating all possible complete partitionings, since many optimal subsolutions are shared among distinct subproblems. For example, the subproblem of partitioning $n = P - 1$ elements into $P - 2$ partitions arises during the construction of all sub-partitionings of length $P - 1$. This observation is summarized in the following corollary.

Corollary 6.3 Overlapping subproblems. *The sub-partitionings of length $P - 1$ considered by the procedure of Corollary 6.2 are based on optimal solutions to subproblems of length $P - 2$. Many of these subproblems are* identical *for* distinct *sub-partitionings. Thus, if we store these subsolutions, they do not need to be recomputed. It follows that the procedure of Corollary 6.2 is strictly more efficient than exhaustive enumeration of all partitionings.*

The properties (optimal substructure and overlapping subproblems) expressed by Lemma 6.1 and its corollaries are the key properties of problems amenable to dynamic programming. Once these have been formulated, it is often straightfoward to write down an algorithm, which typically involves defining a table to store the subsolutions as noted in Corollary 6.3, and devising an iterative procedure to fill it in. In most cases, this table stores the costs rather than the subsolutions themselves;

the optimal solution associated with the minimum cost can then be recovered from the cost table or *via* auxiliary data structures.

Corollary 6.2 immediately gives rise to a two-dimensional table $T[n, p]$ that stores the cost (4) of the optimal partitioning of n elements into p partitions, as well as a way of filling it in:

$$T[n,p] \leftarrow \min_{l=p-1,\ldots,n-1} \left\{ T[l,p-1] + C_{l+1,n}^2 \right\} \tag{10}$$

To turn this into a complete algorithm, all we need is a starting configuration and an order in which the table cells are to be filled; these are given in Table 2. To show that this algorithm correctly produces a globally optimal solution, there is nothing left to prove, as this follows immediately from Corollary 6.2. The only issue that requires a word of explanation concerns the bounds on n in the loops on lines 2 and 5. These arise from the observation that no optimal solution leaves any day without a reading; it is thus useless to fill in any table cells with $n < p$ or $N - n < P - p$.

Having completed the table T, the cost of the minimal solution is located at $T[N, P]$. But how do we recover the actual solution? One way is to remember, in an auxiliary table $L[n,p]$, the value of l that minimized the cost stored at $T[n,p]$ (10). Thus, $L[n,p]$ contains the index of the final chapter assigned to day $p - 1$. We now know that the last day's readings comprise the chapters $L[N, P] + 1, \ldots, N$. To obtain the preceding readings, it suffices to go back to $L[L[N, P],P - 1]$ and iterate. Even better, to produce the daily readings in the correct order, use head recursion instead of iteration.

Example 6.4 Consider $N = 6$ chapters of weights $w_{n=1,\ldots,6} = 2, 5, 3, 4, 7, 6$, and $P = 3$. This yields

$$T[n,p] = \begin{bmatrix} 4 & & \\ 49 & 29 & \\ \mathbf{100} & 58 & 38 \\ 196 & 98 & 74 \\ & \mathbf{221} & 147 \\ & & \mathbf{257} \end{bmatrix}, L[n,p] = \begin{bmatrix} 0 & & \\ 0 & 1 & \\ \mathbf{0} & 2 & 2 \\ 0 & 2 & 3 \\ & \mathbf{3} & 4 \\ & & \mathbf{5} \end{bmatrix},$$

where, following matrix conventions, the first and second indices refer to rows and columns, respectively, and untouched cells are left blank. Bold-face numbers are part of the globally optimal solution, which is:

Table 2. A dynamic-programming algorithm.

1:	**procedure** PLAN
2:	**for** $n = 1, \ldots, N - P + 1$ **do**
3:	$T[n, 1] = C_{1,n}^2$
4:	**for** $p = 2, \ldots, P$ **do**
5:	**for** $n = p, \ldots, N - P + p$ **do**
6:	$T[n,p] \leftarrow \min_{l=p-1,\ldots,n-1} \left\{ T[l,p-1] + C_{l+1,n}^2 \right\}$ (10)

- Day 1: chapters 1–3, $C_{1,3} = 10$
- Day 2: chapters 4–5, $C_{4,5} = 11$
- Day 3: chapter 6, $C_{6,6} = 6$

The running time of our algorithm is dominated by Equation (10) on line 6. For a given table cell this takes $O(N - P)$ time to compute, and there are $O((N - P)P)$ cells to fill, so the complete algorithm takes $O((N - P)^2 N)$ time. As Example 6.4 illustrates, allocation of untouched cells in tables T and L can be avoided by clever indexing, so the algorithm requires $O((N - P)P)$ space.

Some minor improvements are apparent, but they are without any consequences for the asymptotic resource requirements. First, $T[n,P]$ and $L[n,P]$ do not need to be computed for $n < N$. Then, since Equation (10) only ever refers back to column $p - 1$, table T can be collapsed into two columns, one to hold the costs of day $p - 1$ and the other for the costs of the current day p. However, table L cannot be collapsed in this way, since up to the very last day the final, globally optimal partitionings for the first days remain unknown.

7. A graph-based algorithm

Another popular approach to solving a problem involves its formulation as a standard problem on a graph $G = (V,E)$ such that it can be solved *via* one of the many well-understood algorithms for graph problems. For our reading planner, one straightforward idea is to map it onto a shortest-path problem, where edges $e \in E$ in the graph correspond to daily readings, and vertices $v \in V$ correspond to the total number of chapters covered up to a given day. The resulting graph corresponding to Example 6.4 is illustrated in Figure 1.

In general,

$$V = \{n = 0, \ldots, N\} \times \{p = 0, \ldots, P\}$$
$$E = \{(v_{m,p}, v_{n,p+1}) \mid m < n \leq N\}$$
$$w(v_{m,p}, v_{n,p+1}) = C^2_{m+1,n}$$

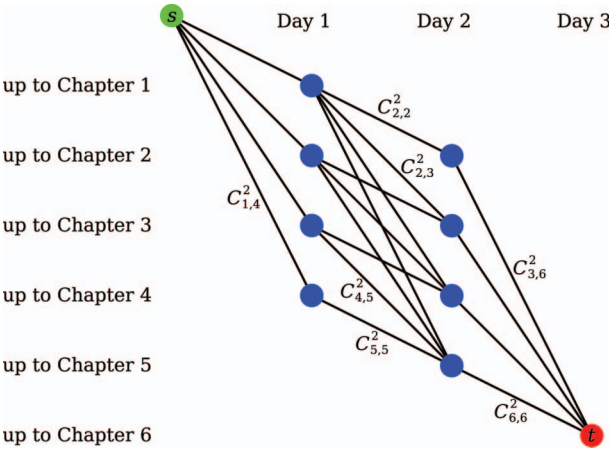

Figure 1. The graph corresponding to Example 6.4, with some of its edge weights.

where $w(u,v)$ is the weight of edge (u,v). From this graph, one may omit any vertices (and their incident edges) that are part of solutions that leave one or more days without any reading, as was done in Figure 1. Then, an optimal reading plan is clearly given by the shortest path from vertex $s = v_{0,0}$ to vertex $t = v_{N,P}$.

To analyze the running time, observe that G contains (in its trimmed version) $O((N-P)P)$ vertices and $O((N-P)^2)$ edges per day, i.e. $O((N-P)^2N)$ edges in total. Using an array of cumulative sums of the w_i, each edge weight can be computed in constant time. Thus, the time to construct the graph is $O((N-P)^2N)$.

If Dijkstra's algorithm is used to find the shortest path, its running time depends on the implementation of the underlying priority queue used to find the 'closest' vertex to be added at each iteration:

- With a binary heap, Dijkstra's algorithm runs in $O(|E|\log|V|)$, which is $O((N-P)^2N\log((N-P)P))$ in our case.
- With a Fibonacci heap, this can be reduced to $O(|E| + |V|\log|V|) = O((N-P)^2N + (N-P)P\log((N-P)P))$, which is $O((N-P)^2N)$ for $N \gg P$.

Thus, the best guarantee we can make for the asymptotic running time of our graph-based solution is $O((N-P)^2N)$ for $N \gg P$, which arises due to the dense nature of our graph. By the same argument, its space requirements are also $O((N-P)^2N)$.

Comparing Figure 1 to table T of Example 6.4, some similarities between our graph and dynamic-programming algorithms are apparent. The idea of tracing separators over days is explicit in the graph-based algorithm, and shows up in reverse direction during the recursive traversal of L. Both algorithms turn out to have the same asymptotic running time. However, the space requirements of the graph-based algorithm are greater because all possible daily readings are represented explicitly, while the tables of the dynamic-programming algorithm only represent the equivalent of the vertices.

From a practical point of view, the graph-based algorithm was easy to motivate, but its implementation draws on some rather sophisticated machinery. It requires a graph data structure and an implementation of Dijkstra's algorithm, which internally uses a Fibonacci heap. The latter is sufficiently complex to make it difficult to cover within an undergraduate curriculum. By contrast, the dynamic-programming algorithm is rather trivial to implement; all we need is two tables and two nested loops (plus a third to take the minimum on line 6). This is typical of dynamic-programming algorithms: Discovering a useful optimal substructure that allows sharing of subproblems can be a challenge – here this gave rise to the longest section of this article –, but once the idea stands, it is often a matter of a few lines to write down and even implement the algorithm.

8. Work related to the problem of planning readings

The problem studied in this article appears to be original. However, it is quite similar to the problem of chains-on-chains partitioning that has been extensively studied (Pınar & Aykanat, 2004). As opposed to balancing the load among partitions under a squared cost function, the chains-on-chains partitioning problem is to minimize the bottleneck load, that is, the maximum cost C_p over

all partitions p. This problem is distinct from the reading planner, as the following example shows.

Example 8.1 Consider $N = 7$ chapters of weights $w_{n=1,\ldots,7} = 1, 4, 1, 2, 1, 2, 2$, and $P = 4$. The minimum-cost (4) solution is $n_{p=1,2,3} = 2, 4, 6$ with $C^2 = 47$. In terms of the chains-on-chains problem, this solution has a bottleneck load of $C_1 = 5$. The solution $n_{p=1,2,3} = 1, 2, 5$ reduces the bottleneck to $C_2 = C_3 = C_4 = 4$, but increases the cost (4) to $C^2 = 49$.

Intuitively, the chains-on-chains problem may be more efficiently solvable than the reading-planner problem, since only a single bottleneck value is to be minimized. In general, many different optimal partitionings will respect the same optimal bottleneck value but will differ in their squared cost (4). Indeed, the asymptotically fastest known algorithms for chains-on-chains are based on an explicit search for the optimal bottleneck value (Nicol, 1994; Pınar & Aykanat, 2004), an idea that is not applicable to the reading-planner problem. Contrary to the results presented in Sec. 7, no competitive graph-based algorithms appear to be known for the chains-on-chains problem (Nicol and O'Hallaron, 1991; Pınar and Aykanat, 2004).

The dynamic-programming algorithm of Section 6 corresponds to the baseline algorithm discussed by Pınar and Aykanat (2004). The authors present a series of improvements; most are based on the idea of bounding the bottleneck value, which is not directly applicable to the reading-planner problem.

9. Role within the curriculum

One fundamental skill of computer scientists is the solution of real-world problems by using standard algorithmic methods. The acquisition of this skill requires two steps, (1) the mastery of the standard methods (knowledge acquisition), and (2) mapping real-world problems onto appropriate methods (knowledge transfer). Most conventional instruction on algorithms focuses on the first objective. The problem and solutions presented in this article are designed to address the second objective by providing a real-world context and adding a transversal, comparative dimension to the conventional (and essential), focused studies.

Thus, this material is not intended for sequential presentation by the instructor. Instead, I recommend that the students be given maximal freedom for brainstorming and exploration. Student contributions can then feed into classroom discussions, covering various algorithms as they come up, in no particular order. Room should be given to discuss variants of algorithms, as such reflections and their results can be quite instructive. For example, in one of my classroom discussions, students tried hard to fix the greedy method, until they discovered why no simple variant of the discussed greedy algorithm can be optimal.

Such engaging, in-class discussions are difficult to entertain productively when introducing new algorithms because most are sufficiently intricate to require careful presentation by a knowledgeable instructor. Once having acquired this new knowledge, a foundation has been laid that permits – and calls for – its consolidation by additional means that involve increased active participation and higher-order reflection for generalization and transfer (Bransford, Brown, & Cocking, 2000, chap. 3). This is the intended use of the material presented in this article.

10. Conclusion

This article presented an original problem designed to foster the exploration of various standard types of algorithms by undergraduate students of computer science. The problem is drawn from daily student life and is easily formalized for rigorous treatment. Although being intuititive and easy to state, it is sufficiently rich to demonstrate the merits of sophisticated types of algorithms for efficient, optimal solutions. Yet, the solutions presented are all relatively straightforward instances of these types of algorithms, and are accessible to the designated audience.

The best of the demonstrated solutions beautifully illustrates the elegance of dynamic programming, an algorithmic paradigm that is among the most challenging to grasp for novice computer scientists. It has interesting parallels to the other optimal algorithm presented, a typical and illustrative instance of a graph-based approach, whose running time is competitive but which is less space efficient and substantially harder to implement.

In terms of the underlying real-world problem, even some of the suboptimal solutions may have their merits. In practice, students may prefer the second greedy algorithm, which is very simple and efficient, and split heavy chapters across days or take a break after demanding readings.

I recommend the reading-planner problem and the solutions presented as a potentially valuable addition to the instructional toolbox of computer science educators, complementing the prevalent (and foundational) algorithm-centric approach by a comparative discussion around a real-life problem.

References

Bransford, J.D., Franks, J.J., Vye, N.J., & Sherwood, R.D. (1989). New approaches to instruction: Because wisdom can't be told. In S. Vosniadou & A. Ortony (Eds.), *Similarity and analogical reasoning* (pp. 470–497). Cambridge, UK: Cambridge University Press.

Bransford, J.D., Brown, A.L., & Cocking, R.R. (Eds.). (2000). *How people learn: Brain, mind, experience, and school* (Expanded Edition). Washington, DC: Commission on Behavioral and Social Sciences and Education, National Research Council, National Academy Press.

Broudy, H.S. (1977). Types of knowledge and purposes in education. In R.C. Anderson, R.J. Spiro, & W.E. Montague (Eds.), *Schooling and the acquisition of knowledge* (pp. 1–17). Hillsdale, NJ: Erlbaum.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Cambridge, MA: MIT Press.

Ericsson, K., Chase, W., & Faloon, S. (1980). Acquisition of a memory skill. *Science*, *208*, 1181–1182.

Goodrich, M.T., & Tamassia, R. (2005). *Data structures and algorithms in Java* (3rd ed.). New York: Wiley.

Nicol, D.M. (1994). Rectilinear partitioning of irregular data parallel computations. *Journal of Parallel and Distributed Computing*, *23*(2), 119–134.

Nicol, D.M., & O'Hallaron, D.R. (1991). Improved algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, *40*(3), 295–306.

Pınar, A., & Aykanat, C. (2004). Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, *64*, 974–996.

Schwartz, D.L., Lin, X., Brophy, S., & Bransford, J.D. (1999). Toward the development of exibly adaptive instructional designs. In C. Reigelut (Ed.), *Instructional design theories and models* (Vol. II, pp. 183–213). Hillsdale, NJ: Erlbaum.

Sedgewick, R. (2003). *Algorithms*. Boston, MA: Addison Wesley Professional.

Appendix 1. Total number of distinct partitionings

Lemma 1.1. Number of distinct solutions. *The number s(N,P) of distinct partitionings is equal to both the recurrence relation (5) and the binomial coefficient (6):*

$$s(N,P) = \begin{cases} 1 & \text{if } P \leq 1, \\ 1 + \sum_{n=P}^{N-1} s(n, P-1) & \text{otherwise.} \end{cases}$$
$$= \binom{N-1}{P-1} \tag{A1}$$

Proof (By induction on N)

Base cases $s(1,P) = 1$ for both the recurrence relation and the binomial coefficient, as is easily verified. This holds true for *any* value of $P > 0$.

Inductive Step Suppose Equation (A1) holds for a given value of $N > 0$ and for any value of $P > 0$. Then, it also holds true for $N + 1$:

$$\begin{aligned} s(N+1,P) &= 1 + \sum_{n=P}^{N} s(n, P-1) \\ &= s(N,P) + s(N, P-1) \\ &= \binom{N-1}{P-1} + \binom{N-1}{P-2} \\ &= \frac{(N-1)!}{(P-1)!(N-P)!} + \frac{(N-1)!}{(P-2)!(N-P+1)!} \\ &= \frac{(N-1)!(N-P+1) + (N-1)!(P-1)}{(P-1)!(N-P+1)!} \\ &= \frac{N!}{(P-1)!(N-P+1)!} \\ &= \binom{N}{P-1} \end{aligned} \tag{A2}$$

where Equation (A2) holds by grounding the inductive hypothesis of the two terms in different base cases.