Algorithm Exercises Skyline and Young Tableau: Divide-and-Conquer Revisited

Gerth Stølting Brodal
Aarhus University, Denmark

Abstract

We revisit the two classic algorithm exercises *skyline* and *Young tableau*, often given to students in an introduction to algorithms course. For both problems we present alternative, still very simple, divide-and-conquer solutions achieving running-times better than what is traditionally asked to achieve in these exercises, in the sense that the running times we achieve are output and input sensitively, respectively. Computing the skyline of a list of n buildings is a classic algorithm problem, solvable with various sweep line and divide-and-conquer approaches in $O(n \lg n)$ time. The classic divide-and-conquer solution resembles mergesort, merging skylines of subsets of the buildings. In this paper we describe an alternative simple divide-and-conquer approach (using Kirkpatrick and Seidel's marriage-before-conquest technique), achieving an optimal output sensitive running time of $O(n \lg k)$, where k is the number of buildings contributing to the skyline. For the Young tableau problem, we consider searching rectangular $m \times n$ matrices which are both row and column monotone, and where the original problem asks to find an algorithm with running time $O(m+n)$. We present a simple divide-and-conquer algorithm for searching matrices in worst-case optimal running time $O\big(m\big(1+\lg \frac{n}{m}\big)\big)$, where $m \leq n$. We also present an input sensitive search algorithm with optimal running time $O\big(k\big(1+\lg \frac{n}{k}\big)\big)$, where k is the complexity of the boundary in the matrix between values smaller and larger than the query value (k is the number of vertical line segments on the boundary, where $k \leq m$). Here optimality refers to the best possible running time when expressing the running time in terms of m and n, or m, n and k, respectively.

2012 ACM Subject Classification Theory of computation → Divide and conquer

Keywords and phrases Divide and conquer, marriage before conquest, output-sensitive running time

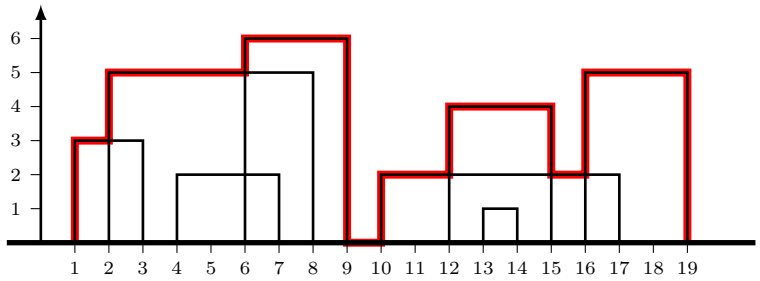
Digital Object Identifier 10.4230/LIPIcs.ESA.2026.43

1 Introduction

A good algorithm exercise/problem should be simple to state. At the same time it should allow students to grow by finding solutions of varying complexity by exploiting different ideas. An example is searching for a value in a sorted list of n values, that canonically leads to a binary search accessing $1 + \lfloor \lg n \rfloor$ values of the list,[1] where the next step is to prove the optimality of binary search by an adversary argument. Generalizing the search to an *unbounded* sorted list asks the students to come up with exponential search followed by a binary search, totally accessing $1 + 2\lfloor \lg p \rfloor$ values, where p is the position of the value searched for in the list (or its successor if not present). Bentley and Yao [2] gave a sequence of algorithms performing better for unbounded search, ultimately achieving an algorithm accessing $\sum_{i=1}^{\lg^* p} \lg^{(i)} p + O(\lg^* p)$ values and proved that this is optimal up to an $O(\lg^* p)$ term.[2]

In this paper we revisit two classic algorithm problems, namely the *skyline* problem, where the task is to compute the skyline (silhouette) of a set of buildings, and the *Young tableau* problem, where we consider searching matrices that are both row and column monotone.

[1] $\lg n$ denotes the binary logarithm of n.
[2] $\lg^{(1)} p = \lg p$, $\lg^{(i)} p = \lg(\lg^{(i-1)} p)$ for $i > 1$, and $\lg^* p = \min\{i \mid \lg^{(i)} p \leq 1\}$.



Buildings: $[(1,3,3),(2,5,8),(4,2,7),(6,6,9),(10,2,17),(12,4,15),(13,1,14),(16,5,19)]$
Skyline: $(1,3,2,5,6,6,9,0,10,2,12,4,15,2,16,5,19)$

Figure 1 The skyline (thick red line) of eight buildings. Blue values are heights.

For both problems, we present alternative solutions achieving improved output- and input-sensitive running times, using simple divide-and-conquer ideas traditionally already taught in the context where these questions are given to the students.

1.1 The skyline problem

When teaching the algorithm design technique divide-and-conquer, a very popular problem is to ask the students to solve the skyline problem using divide-and-conquer. The input to the problem is a list of n buildings, each represented by a triple (ℓ, h, r), where $\ell < r$ are the x-coordinates of the left and right corners of the building, respectively, and $h > 0$ is the height of the building. The output is the polyline capturing the skyline (silhouette) of the buildings, represented as $(x_1, h_1, x_2, h_2, \ldots, x_{k-1}, h_{k-1}, x_k)$, where $x_1 < x_2 < \cdots < x_k$ are the x-coordinates of the corners of the skyline, and h_i is the height of the skyline/highest building between x_i and x_{i+1}. Note that a single building is a skyline by it self. See Figure 1.

The origin of the skyline problem is unclear to the author of this paper. But it is a popular problem on various programming contest and coding interview websites, including GeeksforGeeks[3] and LeetCode[4], and numerous $O(n \lg n)$ time solutions to the problem can be found on the internet (with slight variations of the output format). The problem can be solved in many ways, e.g., by a horizontal sweep left-to-right keeping track of the buildings intersecting the sweepline in a heap, a vertical bottom-up sweep with a balanced binary search tree with the skyline of the buildings below the sweepline, or a recursive divide-and-conquer algorithm merging skylines of subsets of the buildings.

The goal of this paper is to revisit the skyline problem, and to show that one can achieve an asymptotic running time better than $O(n \lg n)$, when the number of horizontal segments k on the skyline stemming from the input buildings is small. As illustrated below, in the worst-case $k = 2n - 1$ (left) and in the best-case $k = 1$ (right).

Kirkpatrick and Seidel [10] introduced the variant of the divide-and-conquer technique called marriage-before-conquest, to compute the convex hull of a set of n points in the plane in time $O(n \lg h)$, where h is the number of points on the convex hull. In Section 2 we

[3] www.geeksforgeeks.org/dsa/the-skyline-problem-using-divide-and-conquer-algorithm/
[4] www.leetcode.com/problems/the-skyline-problem/

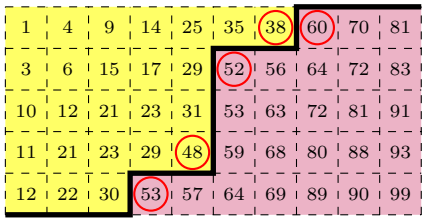


Figure 2 Boundary (solid line) of the search for 50 in a 5×10 Young tableau, where the boundary contains $k = 3$ vertical segments. Values < 50 are yellow and ≥ 50 are red. Circles are entries that any correct algorithm must at least access to correctly determine that 50 is not in the Young tableau.

apply the marriage-before-conquest technique to the skyline problem to achieve the improved running time of $O(n \lg k)$ (which matches a known upper bound by Nielsen and Yvinec [13], see discussion below).

The problem of computing the skyline of a set of buildings is actually just the problem of computing the upper envelope of a set of horizontal line segments, which is a special case of the more general problem of computing the upper envelope of a set of arbitrary line segments in the plane. Whereas the output complexity of the upper envelope of n horizontal line segments or n non-intersecting (not necessarily horizontal) segments contain at most $2n - 1$ segments of the input lines, the general problem where line segments can intersect can have output complexity $\Theta(n \cdot \alpha(n))$, where α is the functional inverse of Ackermann's function [7, 14]. Atallah [1] and Hart and Sharir [7] showed how to compute the upper envelope of n arbitrary line segments, possibly intersecting, in $O(n \cdot \alpha(n) \cdot \lg n)$ time. Hersberger [8] improved this to $O(n \lg n)$ time, and Nielsen and Yvinec [13] further improved this to optimal $O(n \lg k)$ time, where k is the number of segments on the upper envelope. Since the skyline problem is a very special case of the general upper envelope problem, our algorithm is much simpler than the one in [13].

Interestingly, in an educational context students are often asked to solve the skyline problem using divide-and-conquer to illustrate this algorithm design technique. Our algorithm is also purely based on divide-and-conquer. We divide the rectangles on the x-coordinate by using the linear time selection by Blum et al. [3] – that is another textbook example of a (nontrivial) usage of divide-and-conquer, see, e.g., [4, Chapter 9.3] (which can be considered an application of the related prune-and-search technique by Megiddo [11]). We then recursively solve the problem for vertical slabs, while *pruning* buildings that cannot contribute to the skyline in the slabs before recursing.

1.2 The Young tableau problem

Spoiler alert. This paper contains solutions to Problem 6-3(f) in Cormen et al. [4]:

> *f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.*

Cormen et al. [4, Problem 6.3] denote an $m \times n$ matrix a Young tableau, if all rows and columns are sorted in non-decreasing order. Their exercise asks students to use a Young tableau as a data structure to support insertions, extract-min and searches for values in $O(m + n)$ time, where ∞ entries are considered empty. In this context a Young tableau can be viewed as a generalization of the heap order of a binary heap [6, 15] to a two-dimensional grid. In this paper we only consider the searching operation.

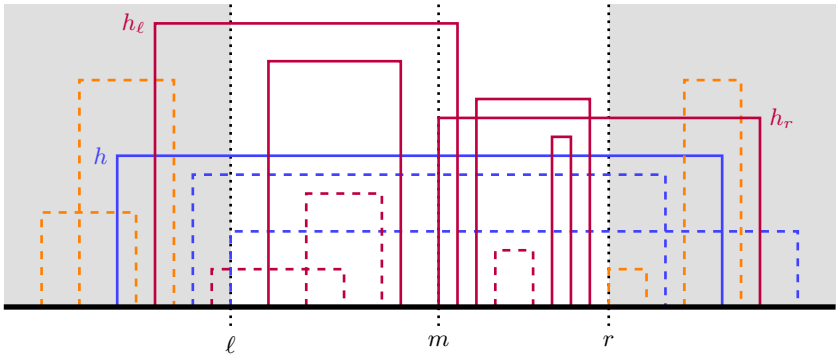


Figure 3 Pruning buildings that cannot contribute to the skyline in the slab $[\ell, r]$.

We present a simple divide-and-conquer algorithm for searching a Young tableau in optimal worst-case $O\big(m\big(1+\lg\frac{n}{m}\big)\big)$ time, where $m \leq n$ (Section 3.3). We also present two different input sensitive algorithms with optimal worst-case $O\big(k\big(1+\lg\frac{n}{k}\big)\big)$ time (Sections 3.4 and 3.5), where k is the complexity of the boundary between the values in the matrix smaller and larger than the value searched for (k is the number of vertical line segments on the boundary, where $k \leq m$). Here optimality refers to the best possible running time when expressing the running time in terms of m and n, or m, n and k, respectively. Figure 2 shows the boundary of a search in Young tableau. Again, our algorithms are applications of algorithm concepts already taught in the context where the Young tableau problem is traditionally stated to students.

2 Output-sensitive skyline algorithm

In this section we present an algorithm for the skyline problem with an improved running time of $O(n \lg k)$ for the case where the resulting skyline is small. Here k denotes the number of buildings contributing to the skyline. Pseudocode for the algorithm is shown in Figure 4.

The idea is to use a divide-and-conquer approach on the x-axis (without sorting all building endpoints). Assume we are going to construct the part of the skyline in the vertical slab $[\ell, r]$ between two x-coordinates ℓ and r (initially between the leftmost and rightmost building corners). Crucial to the algorithm is the *pruning* of buildings that cannot be on the skyline, and therefore can be omitted from consideration in recursive calls. This is the application of the marriage-before-conquest idea of Kirkpatrick and Seidel. See Figure 3. Let h be the height of the highest building spanning the slab ($h = 0$ if no building spans the slab). We prune all buildings (dashed) not intersecting the slab (orange), including those only touching at the endpoints, all buildings spanning the slab below h (blue), and all buildings with one or two top corners with height equal to or below h in the slab (purple). The remaining buildings (solid purple) are the only buildings that could contribute to the skyline in the slab. If there is at least one building remaining, it is crucial for the analysis that there is at least one corner of an input building on the skyline within the slab.

If there is no building left after pruning, the skyline in the slab is simply a horizontal line at height h from ℓ to r. Otherwise, we select the median x-coordinate m of all remaining building corners strictly between ℓ and r in linear time in the number of buildings using the selection algorithm by Blum et al. [3], and recursively construct the skyline for the two slabs $[\ell, m]$ and $[m, r]$ using the pruned set of buildings. In the example in Figure 3, in the slab $[\ell, m]$ all buildings are pruned by the building of height h_ℓ spanning the slab, and we do not

```
function Skyline(B)
    // B is a list of buildings [(l_1, h_1, r_1), ..., (l_n, h_n, r_n)]
    ℓ = min{ℓ_i | (ℓ_i, h_i, r_i) ∈ B}
    r = max{r_i | (ℓ_i, h_i, r_i) ∈ B}
    return SkylineSlab(ℓ, 0, r, B)

function SkylineSlab(ℓ, h, r, B)
    // Construct the skyline of the buildings B for the slab [ℓ, r], assuming
    // there is a height h building in the original input whose span includes [ℓ, r]
    h = max({h} ∪ {h_i | (ℓ_i, h_i, r_i) ∈ B ∧ ℓ_i ≤ ℓ ∧ r ≤ r_i})   // max height spanning slab
    B = {(ℓ_i, h_i, r_i) ∈ B | h_i > h ∧ (ℓ < ℓ_i < r ∨ ℓ < r_i < r)}   // prune buildings
    if B = ∅ then
        return (ℓ, h, r)
    else
        m = Median({ℓ_i, r_i | (ℓ_i, h_i, r_i) ∈ B})
        L = SkylineSlab(ℓ, h, m, B)
        R = SkylineSlab(m, h, r, B)
        return SkylineConcatenate(L, R)
```

Figure 4 Divide-and-conquer algorithm to compute the skyline of a set of buildings B.

need to recurse within the left slab $[\ell, m]$. In the right slab $[m, r]$, the new spanning building of height h_r prunes one additional building with both corners inside the slab, and we recurse within the right slab on the two buildings with corners strictly above h_r and within $[m, r]$.

The skylines L and R found within the slabs $[\ell, m]$ and $[m, r]$, respectively, might contain adjacent segments at m with the same height (in Figure 3 the top of the building with height h_ℓ is reported as several smaller segments). We clean this before concatenating L and R: Since the rightmost x-coordinate in L equals the leftmost x-coordinate in R, we first remove the rightmost x-coordinate from L. If the rightmost height of L equals the leftmost height of R, we further remove the first x-coordinate and height from R before concatenating L and R (we assume this cleanup is performed by the function SkylineConcatenate in Figure 4).

The resulting divide-and-conquer algorithm is shown in Figure 4. The function SkylineSlab takes as arguments the left and right x-coordinates ℓ and r of the slab, the height h of the highest building spanning the slab that has been pruned, and the remaining buildings B to consider in the slab. The initial call takes as ℓ and r the leftmost and rightmost x-coordinates of all buildings, and as B the complete set of buildings without any pruning, i.e., with $h = 0$.

2.1 Analysis

For the analysis, consider the x-coordinates of the k corner points of input buildings on the found skyline. As observed above, we only need to consider recursive calls on slabs containing at least one of these k x-coordinates strictly within the slab, since the remaining slabs must either not overlap any building or there is a building spanning the slab and having the maximum height of any building overlapping the slab, and therefore all buildings in the slab are pruned. It follows that there are at most k calls at each depth of the recursion with at least one building remaining after pruning. At depth d of the recursion, the number of input x-coordinates in the interior of each slab is at most $2n/2^d$, i.e., in total at depth d there are at most $\min\left(2n, k \cdot 2n/2^d\right)$ such coordinates, since $k \leq 2n$. It follows that the total

size of all subproblems (number of buildings in B), where not all buildings are pruned, is $O\left(\sum_{d=0}^{\lceil \lg n \rceil} \min\left(2n, k \cdot 2n/2^d\right)\right) = O\left(n \lg k + n \sum_{d=\lceil \lg k \rceil}^{\lceil \lg n \rceil} k/2^d\right) = O(n \lg k + n) = O(n \lg k)$. The method SkylineSlab recurses on exactly these subproblems plus one additional level of recursion to realize that all buildings in a slab have been pruned. This at most doubles the total size of the subproblems considered, and it follows that the total running time is $O(n \lg k)$.

2.2 Lower bound

That $O(n \lg k)$ is the best possible output sensitive running time for the skyline problem in the comparison model can be shown by reducing the problem of counting the number of distinct values in a set of n values to computing a skyline.

Assume we have an input of n values, possibly with duplicates. Find the largest value $\bar{x}$ by scanning over the values in $O(n)$ time and performing $n-1$ comparisons. In additional $O(n)$ time and $n-1$ comparisons remove all occurrences of $\bar{x}$ from the set. Let $x_1, x_2, \ldots, x_m$ denote the remaining values (all strictly less that $\bar{x}$), where $0 \leq m < n$. Compute now the skyline of the m buildings $(x_1, x_1, \bar{x}), \ldots, (x_m, x_m, \bar{x})$ (see Figure 5). If the skyline contains $n-1$ heights, then all n input values are distinct, and appear in sorted order on the found skyline. For algorithms only performing comparisons this implies a worst-case $\Omega(n \lg n)$ time lower bound for constructing the skyline by the lower bound for comparison-based sorting. More generally, if the skyline contains $k-1$ heights, then there are k distinct values among the n input values. Below we argue that to determine if there are exactly k distinct values in an input, for $1 \leq k \leq n$, requires worst-case $\Omega(n \lg k)$ comparisons. It follows that computing skylines of size k requires worst-case $\Omega(n \lg k)$ coordinate comparisons.

If a decision tree correctly decides if an input of n values contains exactly k distinct values, $1 \leq k \leq n$, then the comparisons performed by the decision tree on an input with k distinct values must identify the partition of the n values into k non-empty subsets, and determine the relative order of the k distinct values. Therefore, the decision tree must have at least $S(n,k) \cdot k!$ leaves, where $S(n,k)$ is the Stirling number of the second kind counting the number of ways to partition a set of n values into k non-empty subsets, and the height is at least $\lg(S(n,k) \cdot k!) = \Theta(n \lg k)$, since $k^{n-k} \leq S(n,k) \leq k^n/k!$. Altenatively, the lower bound for sorting multi-sets by Munro and Spira [12] implies a lower bound of $\Theta\left(\sum_{i=1}^{k} n_i \lg \frac{n}{n_i}\right)$, where n_i is the multiplicity of the i-th distinct value, i.e., a matching $\Theta(n \lg k)$ lower bound when all $n_i = n/k$.

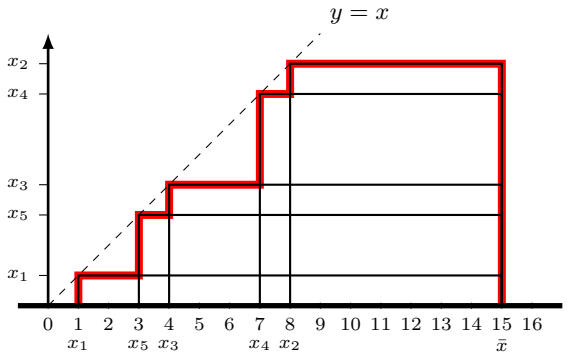


Figure 5 Reduction of values 1, 8, 4, 7, 3, 15 to a set of buildings and their skyline.

3 Searching Young tableaus

In this section we consider the problem of searching for a value x in an $m \times n$ *Young tableau* M, i.e., a matrix M where the values in each row and each column are sorted in non-decreasing order [4, Problem 6.3]. We first recall the standard $O(m+n)$ time search (Section 3.1), and then present an algorithm with improved running time for non-squared matrices (Section 3.3), and algorithms with input sensitive running times (Sections 3.4–3.5). Matching lower bounds are given in (Section 3.2).

3.1 "Textbook" algorithm and lower bound

The basic observation is that if $M_{i,j} < x$, then $M_{i',j'} < x$ for $1 \leq i' \leq i$ and $1 \leq j' \leq j$, since columns and rows are sorted. Similarly, if $M_{i,j} > x$, then $M_{i',j'} > x$ for $i \leq i' \leq m$ and $j \leq j' \leq n$. This implies, if the lower-left entry $M_{m,1} < x$ then we can prune the first column of M from further consideration, and if $M_{m,1} > x$ then we can prune the last row and recurse on the remaining matrix. This gives rise to a simple $O(m+n)$ time algorithm: Start in the lower-left corner $M_{m,1}$ of the matrix, and repeatedly compare x with the values in the row left-to-right until the first value $\geq x$ is found. Either we have found x, or we prune the remaining of the row (since all remaining values in the row are $> x$) and all columns to the left (only containing values $< x$), and we move to the entry in the row above and repeat until either x is found or the whole matrix has been pruned. See Figure 6 (left), where the yellow and red entries are the entries accessed by a search. Since we can move to the right at most $n-1$ times and move up at most $m-1$ times when starting in the lower-left corner, we access at most $1+(n-1)+(m-1) = m+n-1$ entries of M during a search, and we have the following theorem.

▶ **Lemma 1.** *Searching an $m \times n$ Young tableau M for a value can be done in $O(m+n)$ time accessing at most $m+n-1$ entries of M.*

A simple adversary argument shows that this is the best possible worst-case upper bound for $n \times n$ *square* Young tableaus.

▶ **Lemma 2.** *Searching an $n \times n$ Young tableau M for a value requires worst-case $\geq 2n-1$ accesses to M.*

Proof. Consider the adversary strategy illustrated in Figure 6 (middle), where the adversary always answers accesses on or above the anti-diagonal to be $< x$ (say $-\infty$) and accesses below the anti-diagonal to be $> x$ (say ∞). Our claim is that any correct algorithm must access all nodes on the anti-diagonal and the entries immediately below (the colored cells in Figure 2 (middle)). If an algorithm does not access all these $2n-1$ entries, then any of these entries could potentially contain x since this would be consistent with all other entries containing $-\infty$ and ∞ as described above. ◀

The bound of Lemma 1 is obviously not optimal in general, since, e.g., for a $1 \times n$ matrix (a sorted list) we can perform a binary search with $1 + \lfloor \lg n \rfloor$ accesses to the matrix, and this is worst-case optimal for searching a sorted list. In the following sections we present worst-case optimal divide-and-conquer algorithms for searching rectangular $m \times n$ Young tableaus.

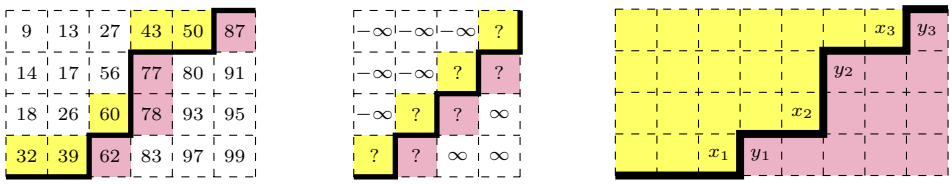


Figure 6 (left) Linear search for 61 in a Young tableau; (middle) $2n-1$ adversary lower bound for square matrices; and (right) generalization to rectangular matrices. Yellow entries are $< x$, red entries $> x$, and the thick line the boundary between larger and smaller values.

3.2 Lower bounds for searching rectangular Young tableau

We first prove a lower bound on the complexity for searching an $m \times n$ Young tableau in terms of the dimensions m and n.

▶ **Lemma 3.** *Searching an $m \times n$ Young tableau M, where $m \leq n$, requires worst-case $\Omega\big(m\big(1+\lg \frac{n}{m}\big)\big)$ accesses to M.*

Proof. To prove a lower bound for searching rectangular $m \times n$ Young tableaus, we assume that the algorithm repeatedly accesses the entries of the matrix. For each access the algorithm learns that the entry is either $-\infty$ or ∞. To correctly determine that x is not in the matrix, the algorithm must access sufficiently many entries to have determined which entries are $-\infty$ and which are ∞ in the matrix, i.e., it must have determined the boundary between the $-\infty$ and ∞ entries. The number of possible boundaries from the lower-left corner to the upper-right corner in an $m \times n$ Young tableau is $\binom{m+n}{m}$, since each boundary can be described by a path of length $m+n$ where exactly m segments are vertical (and n horizontal), i.e., each binary string of length $m+n$ with exactly m ones corresponds to a unique possible boundary in a Young tableau.

By considering an algorithm to be a decision tree where each node accesses an entry of the matrix, the two children correspond to the two possible values $-\infty$ and ∞, and each leaf is labelled with the determined boundary, it follows that the height of the decision tree must be at least $\left\lceil \lg \binom{m+n}{m} \right\rceil = \Theta\big(m\big(1+\lg \frac{n}{m}\big)\big)$. ◀

We can tighten the lower bound of Lemma 3 by observing that the boundary between values smaller and larger than the value searched for can consist of $k \leq m$ longer vertical segments. See, e.g., Figure 6 (right), where a possible boundary in a 4×8 Young tableau consists of $k = 3$ vertical segments. Note that an algorithm in this example must have accessed the $2k$ lower-right and upper-left "corner" entries x_1, x_2, x_3 and y_1, y_2, y_3 in "smaller" and "larger" regions, respectively, to correctly have determined this boundary. In general, x_1 and y_k might not exist (when the first and last vertical segments are on the boundary of the matrix). It follows that any algorithm must have accessed at least between $2k-2$ and $2k$ entries, depending on the boundary. The following lemma gives a more fine-grained lower bound.

▶ **Lemma 4.** *Searching an $m \times n$ Young tableau M, where $m \leq n$, and the boundary of the search contains k vertical segments, requires worst-case $\Omega\big(k\big(1+\lg \frac{n}{k}\big)\big)$ accesses to M.*

Proof. To count the number of possible boundaries with k vertical segments, each of the k vertical segments can be to the left of the first column, between two columns or to the right of the last column, i.e., the vertical segments can be at $n+1$ horizontal positions. The first vertical segment starts at the bottom and the last vertical segment ends at the

top of the matrix, but the remaining vertical positions for the start and end of segments are $k-1$ vertical positions out of $m-1$ possible ones. It follows that the total number of boundaries with k vertical segments is $\binom{n+1}{k} \cdot \binom{m-1}{k-1}$. From the discussion before the lemma, we already have a lower bound of $2k-2$ accesses, i.e., we get the following worst-case lower bound on the number of accesses for a search $\max\big(2k-2, \lceil \lg\big(\binom{n+1}{k} \cdot \binom{m-1}{k-1}\big)\rceil\big) \geq \frac{1}{2}(2k-2) + \frac{1}{2}\lceil \lg\big(\binom{n+1}{k} \cdot \binom{m-1}{k-1}\big)\rceil \geq k - 1 + \frac{1}{2}\lg\binom{n+1}{k} = \Omega\big(k\big(1+\lg\frac{n}{k}\big)\big)$. ◀

A note on random matrices

Assume we construct an $m \times n$ random Young tableau M as follows: Fill an $m \times n$ matrix with random values chosen independently and uniformly from $[0,1]$, sort the rows, and finally sort the columns. It is easy to verify that the resulting matrix is both row and column sorted. In this Young tableau, the vertical segments of the boundary of the search for a value $x \in [0,1]$ (assuming M is constructed independently from x) are adjacent to columns $xn \pm O\big(\sqrt{n \lg n}\big)$ with high probability. This follows since before the sorting steps, each row contains expected xn values $< x$, and a Chernoff bound [5, Theorem 1.1] implies that the number of values $< x$ in each row is within $xn \pm O(\sqrt{n \lg n})$ with high probability. It follows that each row before sorting columns has at least $xn - O\big(\sqrt{n \lg n}\big)$ elements $< x$ and at least $n - xn - O\big(\sqrt{n \lg n}\big)$ values $\geq x$ with high probability. Sorting columns preserves this statement. It follows that the expected number of vertical segments k on the boundary is $O\big(\sqrt{n \lg n}\big)$, which can be significantly smaller than m for larger m, e.g., when $m = n$.

3.3 Divide-and-conquer search in Young tableaus

A simple divide-and-conquer algorithm for searching an $m \times n$ Young tableau M for a value x is as follows: Perform a binary search in the middle row $i = \lceil m/2 \rceil$ to find the column j such that $M_{i,j} < x$ and $M_{i,j+1} \geq x$, where $0 \leq j \leq n$. If $M_{i,j+1} = x$ we are done. Otherwise, we recursively search for x in the two submatrices $M_{1..i-1,1..j}$ and $M_{i+1..m,j+1..n}$ (we only recurse if a submatrix is nonempty, allowing us to skip searching in some rows). The remaining of the matrix is pruned from the search. See Figure 7 for the pruning after one binary search, and Figure 8 for an example of a recursion over the rows.

Analysis

In the worst case, we perform a binary search in a range in each row of M, which gives a trivial upper bound on the number of accesses to M of $m(1 + \lfloor \lg n \rfloor)$. However, we can give a tighter upper bound.

Observe that the maximal depth of the recursion over the rows is at most $\lfloor \lg m \rfloor$, since at each level of the recursion the number of rows is halved. At each depth of the recursion, the total number of columns in all subproblems is at most n, since the subproblems are disjoint. E.g., at depth 3 in the recursion in Figure 8, we have searches in rows 1, 3, and 15, which search over columns 1–8, 9–12 and 13–15, respectively.

If the t searches at some depth d is over t subranges of $n_1, n_2, \ldots, n_t$ columns, where $\sum_{i=1}^{t} n_i \leq n$, the searches require at most $\sum_{i=1}^{t}(1 + \lfloor \lg n_i \rfloor) \leq \sum_{i=1}^{t}(1 + \lg n_i)$ accesses to M. Since the sum is maximized when all n_i are equal, we get the upper bound $t + t \cdot \lg \frac{n}{t}$. Since this grows as a function of t, and there are at most 2^d subproblems at depth d, the total number of accesses to M at depth d is at most $2^d + 2^d \cdot \lg \frac{n}{2^d}$. Summing over all depths of the recursion, the total number of accesses to M can be upper bounded by $\sum_{d=0}^{\lfloor \lg m \rfloor}\left(2^d + 2^d \cdot \lg \frac{n}{2^d}\right) = O\big(m + m \lg \frac{n}{m}\big)$. It follows that the divide-and-conquer algorithm searches an $m \times n$ Young tableau in $O\big(m\big(1 + \lg \frac{n}{m}\big)\big)$ time, matching the lower bound in Lemma 3.

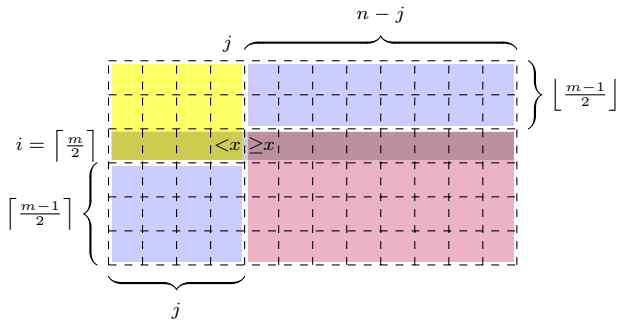


Figure 7 Divide-and-conquer search in a Young tableau for the value x. A binary search in the middle row i (gray) reduces the search to two searches in the two blue submatrices.

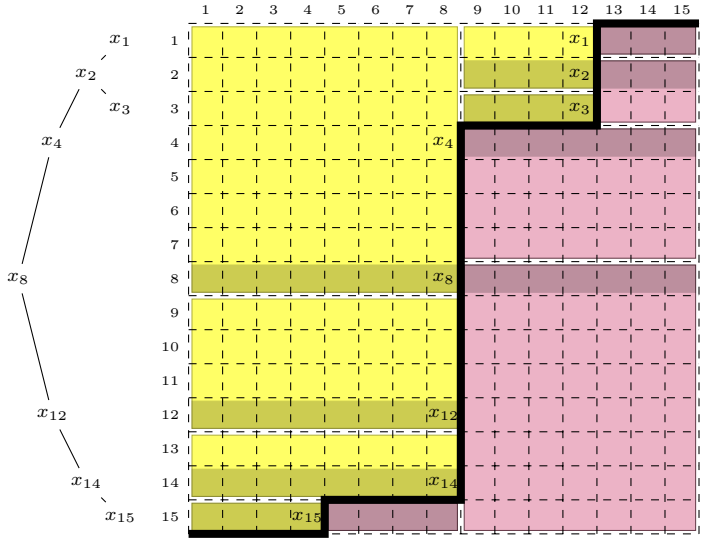


Figure 8 Recursion of a divide-and-conquer search in a Young tableau. To the left the recursion tree over the rows. The gray boxes are the ranges of the binary searches for x, and x_i is the found entry by a binary search in row i. Note that in rows 1 and 4 the search is over columns only containing values $> x$, i.e., the column index found is the column to the left of the first column in the searched range.

3.4 Exponential searching in Young tableaus

The "linear" search algorithm described in Section 3.1 and illustrated in Figure 6 (left) takes inherently $\Theta(m+n)$ time. However, we can improve the search time by alternating to perform an exponential search in rows and columns, essentially following the same path as the "linear" search: Start in the lower-left corner $M_{m,1}$, and perform an exponential search rightwards in the row to find the first value $\geq x$. Either we have found x, or we can proceed with an exponential search upwards in the column to find the first value $\leq x$. Either we have found x, or we continue searching horizontally again.

Analysis

If the boundary identified by the searches consists of k vertical segments, the boundary consists of a total of $k' \leq 2k+1$ vertical and horizontal segments of lengths $s_1, s_2, \ldots, s_{k'}$, where $\sum_{i=1}^{k'} s_i = m+n$. Since each exponential search identifies one segment of the boundary

of size s_i in time $O(1+\lg s_i)$, the total time for the search is $O\big(k'+\sum_{i=1}^{k'}\lg s_i\big)$. Since this sum is maximized when all segments have equal length we get the following upper bound on the search time $O\big(k'+k'\cdot\lg\frac{m+n}{k'}\big)=O\big(k\big(1+\lg\frac{n}{k}\big)\big)$, matching the lower bound of Lemma 4.

3.5 Divide-and-conquer with exponential search in Young tableaus

The divide-and-conquer algorithm in Section 3.3 does not achieve optimal performance when k is small. When all values in the Young tableau are ∞, i.e., $k=1$, the algorithm accesses $\Theta(\lg m\cdot\lg n)$ entries where the optimal bound for $k=1$ is $\Theta(\lg n)$. In this section we show that if the divide-and-conquer algorithm performs bidirectional exponential searches in the rows instead of just binary searches, the algorithm's running time is improved to optimal $O\big(k\big(1+\lg\frac{n}{k}\big)\big)$. It is a little bit surprising that such a minor change to the canonical divide-and-conquer algorithm leads to an algorithm matching the lower bound of Lemma 4 and the complexity of the algorithm in Section 3.4. Since the algorithm in Section 3.4 is inherently sequential, the divide-and-conquer algorithm in this section might be interesting when parallelism is available, since it trivially can achieve parallel time $O(\lg m\cdot\lg n)$ on an EREW PRAM with $\Theta(m/\lg m)$ processors (see, e.g., the textbook by JáJá [9, Section 1.3] for a discussion of PRAM models).

The only change to the divide-and-conquer algorithm in Section 3.3 is that if we are searching in row i in the column range j_1 to j_2, we first compare x with M_{i,j_m}, where $j_m=\lfloor(j_1+j_2)/2\rfloor$ is the middle column in the column range. If $x=M_{i,j_m}$, then we are done. If $x<M_{i,j_m}$, we perform a rightwards exponential search in columns $j_1,\ldots,j_m-1$ (starting in column j_1), and otherwise a leftwards exponential search in columns $j_m+1,\ldots,j_2$ (starting in column j_2).

Analysis

The search in a row within a range of n columns partitions the columns into two subranges of lengths n_1 and n_2, where $0\le n_1\le n$ and $n_1+n_2=n$. If $n_1=0$ or $n_2=0$, the time using exponential search becomes $O(1)$, and there is at most one recursive call on half the number of rows. E.g., in Figure 8 the searches for x_4 and x_{12}, establish the segments of the boundary between x_4 and x_8, and x_8 and x_{12}, respectively. Otherwise, the n columns are split at a new horizontal position into two nonempty subranges of lengths n_1 and n_2. In this case, the time for the exponential search is $O(1+\lg\min(n_1,n_2))$.

Let us consider the recursion tree and how the searches in the rows discover a vertical segment of the boundary (e.g., in Figure 8 the searches in rows 4, 8, 12 and 14 discover the vertical segment of the boundary between columns 8 and 9). The existence of vertical segment is first discovered by some search (row 8) and subsequent recursive searches above and below this row (rows 4, 12, and 14) will discover the extent of the vertical segment. Each of these searches take $O(1)$ time. Furthermore, since we recurse over the rows in a binary tree fashion the interval of rows spanning the vertical segment must decrease by at least a factor two for each new interval above or below the initial row, i.e., a vertical segment of length s can at most be discovered by searches in $O(\lg s)$ rows. If the k vertical segments have length $s_1,\ldots,s_k$, where $\sum_{i=1}^k s_i=m$, it follows that the total number of constant time row searches where either the left or right recursive problem is empty is at most $\sum_{i=1}^k O(1+\lg s_i)=O\big(k\cdot\lg\frac{m}{k}\big)$.

The remaining searches perform a hierarchical decomposition of the columns into at most $k+1$ subranges between vertical segments on the boundary of length $c_1,\ldots,c_{k+1}$, where $\sum_{i=1}^{k+1}c_i=n$. Note that $c_1,\ldots,c_{k+1}$ are the lengths of the horizontal segments on the

boundary. The cost of these searches can be captured by a binary tree with the subranges $c_1, \ldots, c_{k+1}$ as the leaves. If a node has two subtrees spanning n_1 and n_2 columns, respectively, the cost of the search is $O(1 + \lg \min(n_1, n_2))$. To analyze the total cost of these searches, we apply the following amortized analysis. Assume we give each leaf c_i an initial potential of $2+\lg c_i$. We argue inductively bottom-up that the cost of searching in a subtree can be covered by this leaf potential. Assume we have processed the two subtrees of a node with range n and subranges n_1 and n_2, where $n = n_1 + n_2$ and $n_1 \leq n_2$, and there is potential $2 + \lg n_1$ and $2 + \lg n_2$ left for each of the two subtrees, respectively. After using potential $1 + \lg_2 n_1$ for the searching at the node, the potential left is $(2 + \lg n_1) + (2 + \lg n_2) - (1 + \lg n_1) = 3 + \lg n_2 = 2 + \lg(2n_2) \geq 2 + \lg n$, i.e., at the root we have covered all costs for the searches in the binary tree by the cost at the leaves (and we still have $2 + \lg n$ potential left). The total potential required at the leaves is $\sum_{i=1}^{k+1}(2+\lg c_i) \leq 2(k+1)+(k+1)\cdot\lg\frac{n}{k+1} = O\big(k+k\cdot\lg\frac{n}{k}\big)$. In total, the time for searching a Young tableau becomes $O\big(k\cdot\lg\frac{m}{k} + k + k\cdot\lg\frac{n}{k}\big) = O\big(k\big(1+\lg\frac{m}{k}\big)\big)$.

Refined boundary sensitive analysis

The running times of the algorithms discussed in Sections 3.4 and 3.5 have been stated as $O\big(k\big(1+\lg\frac{n}{k}\big)\big)$, which is optimal in terms of k and n by Lemma 4. However, if we consider the boundary to consist of s alternating horizontal and vertical segments of lengths $\ell_1, \ldots, \ell_s$, where $\sum_{i=1}^{s} \ell_i = m + n$, the proofs of the running times of both algorithms actually prove that the worst-case running times are $O\big(s + \sum_{i=1}^{s} \lg \ell_i\big)$.

4 Conclusion and open problems

In this paper we have revisited two classic algorithm exercises: computing the skyline of a set of buildings and searching Young tableaus. For both problems we have presented very simple divide-and-conquer algorithms that are worst-case optimal in an output or input sensitive sense. The analysis of the algorithms become slightly more complex, but we believe that the algorithms and their analyses are still simple enough to be presented as examples of divide-and-conquer in a course on introduction to algorithms, or to be stated as optional questions to illustrate the algorithmic depth of these seemingly innocent undergraduate textbook exercises.

References

1 Mikhail J. Atallah. Some dynamic computational geometry problems. *Computers & Mathematics with Applications*, 11(12):1171–1181, 1985. doi:10.1016/0898-1221(85)90105-1.

2 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.

3 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. URL: https://mitpress.mit.edu/9780262046305/.

5 Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. URL: http://www.cambridge.org/gb/knowledge/isbn/item2327542/.

6 Robert W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701, 1964. doi:10.1145/355588.365103.

7 Sergiu Hart and Micha Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6(2):151–178, 1986. doi:10.1007/BF02579170.

8 John Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, 33(4):169–174, 1989. doi:10.1016/0020-0190(89)90136-1.

9 Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

10 David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal of Computing*, 15(1):287–299, 1986. doi:10.1137/0215021.

11 Nimrod Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal of Computing*, 12(4):759–776, 1983. doi:10.1137/0212052.

12 J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM Journal of Computing*, 5(1):1–8, 1976. doi:10.1137/0205001.

13 Frank Nielsen and Mariette Yvinec. An output-sensitive convex hull algorithm for planar objects. *International Journal of Computational Geometry & Applications*, 8(1):39–65, 1998. doi:10.1142/S0218195998000047.

14 Ady Wiernik and Micha Sharir. Planar realizations of nonlinear Davenport-Schinzel sequences by segments. *Discrete & Computational Geometry*, 3:15–47, 1988. doi:10.1007/BF02187894.

15 J. W. J. Williams. Algorithm 232 – Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. doi:10.1145/363958.363965.