Knuth-Yao Optimization in Dynamic Programming

March 24, 2026

Abstract

Understanding a Dynamic Programming Optimization

***Keywords*.** Dynamic Programming · Algorithms

1 Introduction

Dynamic programming is solving an optimization problem with a given input by composing solutions to the same problem with smaller inputs. The smaller inputs come from breaking down the initial input into smaller pieces, and those pieces into even smaller pieces still, recursively. An important characteristic here is that the solutions to the smaller inputs don't influence each other and the optimal solution to the initial problem has to be composed from optimal solutions to the smaller inputs. Another characteristic is that the composition visits potential smaller solutions over and over again, so it benefits to remember them. This points to the need of having a mapping from input to solution, so an already computed solution can be retrieved instead of recomputed.

In this note we will look at a runtime optimization for a particular set of dynamic programming problems (note here that runtime optimization refers to a speed up in solving a dynamic programming problem which itself is an optimization problem). These runtime optimizations were presented in [1]. This note follows [1] closely and does not contribute anything original, it is purely an exercise in understanding the proofs and adding commentary to them.

In Section 2 we will use an example problem to set the stage. In Section 3 we will follow the proofs from [1] of some inequalities called the *quadrangle inequalities*. These inequalities will be used in Section 4 to reduce the runtime cost of the composition of subproblems. Without the optimization these particular dynamic programming problems have a runtime cost of $O(n^3)$ in the size of the problem, with the optimization the runtime is reduced to $O(n^2)$. In Section 5 we will revisit our example problem to apply the Knuth-Yao Optimization.

2 Problem

The problem in this section is different from the problem used in the introduction of [1] but as we will see, it does lend itself to the same runtime optimization.

Problem. You are given values $x_1, x_2, ..., x_n$. Organize them into a binary tree (without reordering) so that the sum of the values multiplied by their depths in the tree is as small as possible.

Solution. Because the cost formula uses tree depth multiplicatively, we assume that the tree root has depth one.

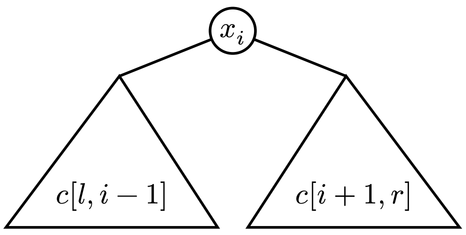
The input is a sorted array of numbers. If the initial input array is globally available then we can operate with intervals as inputs. The interval $[1, n]$ corresponds to the whole array $x_1, x_2, ..., x_n$. A smaller interval $[l, r]$ corresponds to an input array $x_l, ..., x_r$ to our problem.

Breaking down our input interval into smaller intervals would break down the problem into subproblems of the same nature with smaller inputs. Into how many intervals should we break down the input interval? The problem asks for binary trees, so two intervals seems right: left interval $[l, i-1]$ and right interval $[i+1, r]$. The separating position i could be the root of this composition choice. This seems reasonable, so let's flesh it out (at this point we don't know yet if it will work).

Let's denote the smallest possible sum of the values multiplied by their depths as the cost.

We declare $c[l, r]$ as the cost of the binary search tree with nodes $x_l, ..., x_r$.

Our composition suggests the following recursion for $c[l, r]$ and some $i \in [l, r]$ which we assume will give us the smallest sum for input $[l, r]$:



$$\begin{aligned} c[l, r] &= c[l, i-1] + \sum_{k=l}^{i-1} x_k + x_i + c[i+1, r] + \sum_{k=i+1}^{r} x_k \\ &= c[l, i-1] + c[i+1, r] + \sum_{k=l}^{r} x_k \end{aligned} \tag{2.1}$$

If $c[l, i-1]$ was the old cost when it was a tree by itself, now that it is the left subtree of a tree with root x_i, we need to adjust the cost by one level (the tree got pushed down a level and i is its new parent), which means adding $\sum_{k=l}^{i-1} x_k$ to the cost. Similarly for the right subtree $c[i+1, r]$. The sums can be combined to $\sum_{k=l}^{r} x_k$ which does not depend on the choice of i.

We define

$$w[l, r] = \sum_{k=l}^{r} x_k \tag{2.2}$$

Since we don't know upfront which split point i will yield the minimum cost, we have to try them all and choose the minimum from them:

$$c[l, r] = w[l, r] + \min_{l \le i \le r} (c[l, i-1] + c[i+1, r]) \tag{2.3}$$

We define $c[k, k-1] = 0$ for all $1 \le k \le n$.

Is this recursion going to yield the minimum cost? We choose the minimum from the possible roots, so the min expression satisfies one of the requirements of DP: optimal solutions to subproblems are used to compose the solution of the problem. How about $c[l, i-1]$ and $c[i+1, r]$. That sum can be minimized only by the minimum cost of the terms. If there is another binary tree formed for $[l, i-1]$ for example, it has to have minimum cost, otherwise we could swap it out with a tree with lower cost and get a lower total cost. Similarly on the right side.

The recursion does yield the minimum cost.

What is the runtime and space of this solution? The space requirements are $O(n^2)$ since we store $c[l, r]$ in a 2-dimensional array with $1 \leq l \leq r \leq n$.

For runtime we can analyze an implementation that populates the 2-dimensional array going in order of ever increasing interval sizes and for each interval size, sweeping through the $x_1, x_2, ..., x_n$ using a window equal to the interval size.

If we precompute the partial sums $S[k] = \sum_{j=1}^{k} x_j$, for $1 \leq k \leq n$, then we can compute $w[l, r]$ in constant time (assume S[0] = 0): $w[l, r] = S[r] - S[l-1]$.

This means that the expression $c[l, r]$ can be computed in $r + 1 - l$ time.

There are n interval sizes. For a given interval size m we have $n - m + 1$ intervals to process. So runtime is

$$\begin{aligned} \sum_{m=1}^{n}(n-m+1)m &= n\sum_{m=1}^{n} m - \sum_{m=1}^{n} m^2 + \sum_{m=1}^{n} m \\ &= \frac{1}{6}n(1+n)(2+n) = O(n^3) \end{aligned} \tag{2.4}$$

$\square$

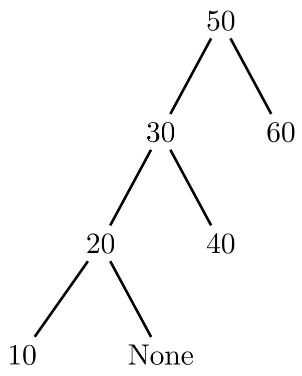


Figure 1: Optimal tree for array $\{10, 20, 30, 40, 50, 60\}$ with cost 450.

3 Quadrangle Inequalities

We define functions on the lattice grid of intervals that satisfy certain inequalities as follows:

Definition 3.1. *A function $w : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that*

$$w(i', j) \leq w(i, j'), \text{if intervals are nested: } [i', j] \subseteq [i, j'] \tag{3.1}$$

*is called monotone on the lattice of intervals, or (in short) a **monotone function**.*

Definition 3.2. *A function $w : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that*

$$\begin{gathered} w(i, j) + w(i', j') \leq w(i', j) + w(i, j'), \\ \text{for } i \leq i' \leq j \leq j' \end{gathered} \tag{3.2}$$

*is said to satisfy the **quadrangle inequality** (QI) and is called a **quadrangle function**.*

These definitions might seem mysterious at first but they do seem vaguely related to (2.3). The domain $\mathbb{N} \times \mathbb{N}$ is an obvious connection to the dynamic programming recursion in the previous section. If we say w is some form of cost, then what the monotone property says is that the cost of a nested interval is smaller than the cost of the enclosing interval.

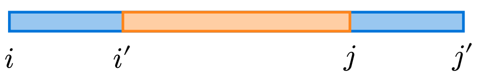


Figure 2: A nested pair of intervals $[i', j] \subseteq [i, j']$ illustrating the monotone condition.

For the quadrangle property, the cost of the overlapping intervals is less than the cost of the nested and enclosed interval.

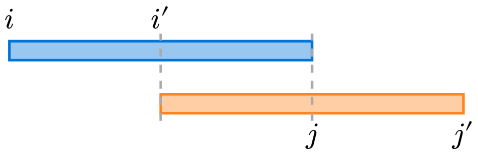


Figure 3: Two overlapping intervals $[i, j]$ and $[i', j']$ with $i \leq i' \leq j \leq j'$, illustrating the quadrangle inequality.

Theorem 3.3. *Given are monotone, quadrangle function* $w : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *and function* $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *defined recursively:*

$$\begin{aligned} c(i,i) &= 0 \\ c(i,j) &= w(i,j) + \min_{i<k\leq j} (c(i,k-1) + c(k,j)) \end{aligned} \tag{3.3}$$

Then function c *is also a quadrangle function.*

Proof. We need to prove

$$\begin{gathered} c(i,j) + c(i',j') \leq c(i',j) + c(i,j'), \\ \text{for } i \leq i' \leq j \leq j' \end{gathered} \tag{3.4}$$

First an observation: if $i = i'$ or $j = j'$ then (3.4) is an equality and trivially satisfied.

We will prove it by induction on $l = j' - i$.

Our base case is $l \leq 1$. When $l = 0$ then all the involved indices are equal and (3.4) is trivially satisfied. With $l = 1$ we are forced into $i = i'$ or $j = j'$ and again (3.4) is trivially satisfied.

For the induction step with lengths $l > 1$ we have two cases (the other cases involve $i = i'$ or $j = j'$ and are dealt with by the observation):

Case A: $i < i' = j < j'$
Case B: $i < i' < j < j'$

For Case A let's rewrite what we need to show:

$$\begin{gathered} c(i,j) + c(j,j') \leq c(j,j) + c(i,j'), \\ \text{for } i \leq j \leq j' \end{gathered} \tag{3.5}$$

We have $c(j,j) = 0$ by definition of c, so the inequality we need to prove simplifies to $c(i,j) + c(j,j') \leq c(i,j')$.

Let z be the splitting point where $c(i,j')$ achieves its minimum, i.e.

$$c(i,j') = w(i,j') + c(i,z-1) + c(z,j') \tag{3.6}$$

We have two cases again, depending in which interval z falls in:

Case A.1: $i < z \le j < j'$
In this case we notice that z is also a splitting point for $c(i,j)$, so it can not be better than the minimum splitting point there:

$$c(i,j) \le w(i,j) + c(i,z-1) + c(z,j) \tag{3.7}$$

We add $c(j,j')$ to both sides and get:

$$c(i,j) + c(j,j') \le w(i,j) + c(i,z-1) + c(z,j) + c(j,j') \tag{3.8}$$

Because w is monotone, we have $w(i,j) \le w(i,j')$, so

$$c(i,j) + c(j,j') \le w(i,j') + c(i,z-1) + c(z,j) + c(j,j') \tag{3.9}$$

We use the induction step on length $j' - z$ and have $c(z,j) + c(j,j') \le c(z,j')$, which gives us:

$$c(i,j) + c(j,j') \le w(i,j') + c(i,z-1) + c(z,j') = c(i,j') \tag{3.10}$$

That takes care of Case A.1.

Case A.2: $i < j < z < j'$
In this case z is also the splitting point for $c(j,j')$, so it cannot be better than the minimum splitting point there:

$$c(j,j') \le w(j,j') + c(j,z-1) + c(z,j') \tag{3.11}$$

We add $c(i,j)$ to both sides and get:

$$c(i,j) + c(j,j') \le w(j,j') + c(j,z-1) + c(z,j') + c(i,j) \tag{3.12}$$

Because w is monotone, we have $w(j,j') \le w(i,j')$, so

$$c(i,j) + c(j,j') \le w(i,j') + c(j,z-1) + c(z,j') + c(i,j) \tag{3.13}$$

We use the induction step on length $z - 1 - i$ and have $c(i,j) + c(j,z-1) \le c(i,z-1)$, which gives us:

$$c(i,j) + c(j,j') \le w(i,j') + c(i,z-1) + c(z,j') = c(i,j') \tag{3.14}$$

This concludes A.2 and thus all subcases of Case A.

We now deal with Case B: $i < i' < j < j'$. As a reminder, we need to prove:

$$\begin{gathered} c(i,j) + c(i',j') \le c(i',j) + c(i,j'), \\ \text{for } i < i' < j < j' \end{gathered} \tag{3.15}$$

We now juggle two splitting points: let y be the minimum splitting point for $c(i',j)$ and again let z be the minimum splitting point for $c(i,j')$.

$$\begin{aligned} c(i',j) &= w(i',j) + c(i',y-1) + c(y,j) \\ c(i,j') &= w(i,j') + c(i,z-1) + c(z,j') \end{aligned} \tag{3.16}$$

Again we have two subcases.
Case B.1: $z \le y$

Splitting point y is also a valid splitting point of $c(i', j')$ so it cannot be better than the minimum splitting point there:

$$c(i', j') \leq w(i', j') + c(i', y-1) + c(y, j') \tag{3.17}$$

Because $z \leq y$, we also have that z is a valid splitting point of $c(i, j)$ so again:

$$c(i, j) \leq w(i, j) + c(i, z-1) + c(z, j) \tag{3.18}$$

Summing up the last two inequalities, we have

$$\begin{aligned} c(i, j) + c(i', j') \leq\ & w(i', j') + c(i', y-1) + c(y, j') \\ & + w(i, j) + c(i, z-1) + c(z, j) \end{aligned} \tag{3.19}$$

Because w is quadrangle, we have

$$w(i, j) + w(i', j') \leq w(i, j') + w(i', j) \tag{3.20}$$

so

$$\begin{aligned} c(i, j) + c(i', j') \leq\ & w(i, j') + w(i', j) + c(i', y-1) + c(y, j') \\ & + c(i, z-1) + c(z, j) \end{aligned} \tag{3.21}$$

By induction step on length $j' - z$ with points $z \leq y \leq j < j'$ we have

$$c(z, j) + c(y, j') \leq c(y, j) + c(z, j') \tag{3.22}$$

which gives us

$$\begin{aligned} c(i, j) + c(i', j') &\leq w(i, j') + w(i', j) + c(i', y-1) + c(y, j) + \\ &\quad c(i, z-1) + c(z, j') \\ &= (w(i, j') + c(i, z-1) + c(z, j')) + \\ &\quad (w(i', j) + c(i', y-1) + c(y, j)) \\ &= c(i, j') + c(i', j) \end{aligned} \tag{3.23}$$

This concludes Case B.1.
Case B.2 has $y \leq z$ and is similar (symmetric) to case B.1, so we won't spell it out. □

In the next section we use Theorem 3.3 to speed up the computation of $c(1, n)$. We have seen in the previous section that the naive computation of $c(1, n)$ takes $O(n^3)$ time.

4 Knuth-Yao Optimization

The goal in this section will be to capture inequalities comparing values of function c as defined in Theorem 3.3. These inequalities will allow us to ignore interval ranges when looking for the minimum index in the expression $c(i, j) = w(i, j) + \min_{i<k\leq j}(c(i, k-1) + c(k, j))$. Discarding some interval ranges when looking for minimum split points will allow us to reduce the runtime.

But before we start, we need more notation:

$$c_k(i, j) = w(i, j) + c(i, k-1) + c(k, j) \tag{4.1}$$

We also define

$$K_c(i, j) = \max\{k \mid c_k(i, j) = c(i, j)\} \tag{4.2}$$

So $K_c(i,j)$ is the largest index k that is a minimum split point for $c(i,j)$. There could be more than one split point that achieve a minimum, so taking the largest lets us argue that any index beyond that would not achieve minimum.

Theorem 4.1.

$$K_c(i,j) \le K_c(i,j+1) \le K_c(i+1,j+1) \tag{4.3}$$

Proof. Let's start with proving $K_c(i,j) \le K_c(i,j+1)$.

What this inequality is saying in plain English: if you extend the right endpoint from j to $j+1$, the largest minimum split point can only move right or stay put. It never moves left. It should be clear how this fact would be useful in a recursive solution: we don't have to search to the left of the smaller interval split point for our next interval split point.

If $i=j$ then $K_c(i,j) \le K_c(i,j+1)$ is trivially true because $c(i,i)=0$. So we assume $i<j$.

Consider the points $i<k\le k'\le j$. We will show that

$$c_{k'}(i,j) \le c_k(i,j) \Rightarrow c_{k'}(i,j+1) \le c_k(i,j+1) \tag{4.4}$$

To show (4.4) we apply the quadrangle inequality to points $k\le k'\le j<j+1$ and get

$$c(k,j)+c(k',j+1) \le c(k',j)+c(k,j+1) \tag{4.5}$$

We add $w(i,j)+w(i,j+1)+c(i,k-1)+c(i,k'-1)$ and by definition (4.1) we get

$$c_k(i,j)+c_{k'}(i,j+1) \le c_{k'}(i,j)+c_k(i,j+1) \tag{4.6}$$

If $c_{k'}(i,j) \le c_k(i,j)$ holds, then $c_{k'}(i,j)-c_k(i,j) \le 0$, and so

$$c_{k'}(i,j+1) \le c_k(i,j+1)+c_{k'}(i,j)-c_k(i,j) \le c_k(i,j+1) \tag{4.7}$$

which proves (4.4). How does $K_c(i,j) \le K_c(i,j+1)$ follow from (4.4) ? Well, we set $k'=K_c(i,j)$, so $c_{k'}(i,j) \le c_k(i,j)$ for any k, so also for k to the left of k'. Implication (4.4) then tells us, that none of these k to the left of k' beat k' in the search for $K_c(i,j+1)$.

The other inequality $K_c(i,j+1) \le K_c(i+1,j+1)$ is proven similarly. □

Our goal is to speed up the computation of $c(1,n)$ where c has the recursive form:

$$\begin{aligned} c(i,i) &= 0 \\ c(i,j) &= w(i,j) + \min_{i<k\le j}(c(i,k-1)+c(k,j)) \end{aligned} \tag{4.8}$$

We use the two-dimensional array $K_c(i,j)$ and two-dimensional array $c(i,j)$.

Theorem 4.1 tells us that $K_c(i,j) \le K_c(i,j+1) \le K_c(i+1,j+1)$ which means the largest minimum split point index values increase when going from (i,j) to $(i+1,j+1)$.

If we populate the arrays by the length of the interval $j-i$, then when we look for $K_c(i,j+1)$ in order to compute $c(i,j+1)$, we only have to consider index values between $K_c(i,j)$ and $K_c(i+1,j+1)$. To find all the K_c of length $m=j+1-i$ we have to visit

$$\sum_{i=1}^{n-m} K_c(i+1,i+m) - K_c(i,i+m-1) \tag{4.9}$$

Let $\sigma(i)=K_c(i,i+m-1)$, then it is visible that we have a telescopic sum

$$\sum_{i=1}^{n-m+1} \sigma(i+1) - \sigma(i) = \sigma(n-m+1) - \sigma(1) \tag{4.10}$$

The total runtime cost is therefore $O(n^2)$.

5 Example Problem revisited

Our example problem had the recursion (2.3). Let us repeat here one more time:

$$c[l,r] = w[l,r] + \min_{l \le i \le r} (c[l,i-1] + c[i+1,r]) \tag{5.1}$$

If we want to apply the Knuth-Yao Optimization to this recursion, we first need to check that w has the required properties: it is a monotone, quadrangle function.

The w in our example problem is

$$w(i,j) = \sum_{k=i}^{j} x_k \tag{5.2}$$

We can assume that the x_k are positive (if they are not, we can shift the whole array up by a positive constant to make it so without disturbing their order).

Then it is trivial to see that w is monotone: the sum of values from the nested interval is smaller than the sum of values of the enclosing interval.

For the quadrangle property we have in fact equality (see Figure 3).

So w is taken care of. Our recursion (2.3) is almost identical to the one in Theorem 3.3 (the left side of the minimum qualifier is inclusive in (2.3) and exclusive in Theorem 3.3). We can fix that by making it exclusive in (2.3) and then doing an additional check when populating the arrays, so it doesn't really hurt the runtime.

This means our example problem can be solved in $O(n^2)$.

6 Summary

It is a beautiful result! The whole paper [1] is really one clean idea:

(i) **QI on w → QI on c**: Theorem 3.3
(ii) **QI on c → monotone split points**: Theorem 4.1
(iii) **Monotone split points → $\mathbf{O(n^2)}$** (the amortization argument (4.10) with telescopic sum)

And the "crossing ≤ nesting" quadrangle inequality (QI) runs through all three steps.

Bibliography

[1] F. F. Yao, "Efficient dynamic programming using quadrangle inequalities," in *Proceedings of the twelfth annual ACM symposium on Theory of computing*, in STOC '80. New York, NY, USA: Association for Computing Machinery, 1980, pp. 429–435. doi: 10.1145/800141.804691.

Appendix A

Go implementation of the optimized solution that also prints out the trees as Cetz tree expressions.

```go
package main

import (
  "flag"
  "fmt"
  "math"
  "strconv"
  "strings"
)

func main() {
  numsStr := flag.String("nums", "10,20,30,40,50,60", "Comma-separated list of numbers")
  flag.Parse()

  var xs []int
  for _, s := range strings.Split(*numsStr, ",") {
    s = strings.TrimSpace(s)
    if s == "" {
      continue
    }
    num, err := strconv.Atoi(s)
    if err != nil {
      fmt.Printf("Invalid number: %q\n", s)
      return
    }
    xs = append(xs, num)
  }

  if len(xs) == 0 {
    fmt.Println("No input numbers provided.")
    return
  }

  n := len(xs)
  dp := make([][]int, n+1)
  for i := range n + 1 {
    dp[i] = make([]int, n+1)
  }
  ip := make([][]int, n+1)
  for i := range n + 1 {
    ip[i] = make([]int, n+1)
  }

  sums := make([]int, n)
  for i := range n {
    dp[i][i+1] = xs[i]
    ip[i][i+1] = i // only valid split for a single-element interval
    sums[i] = xs[i]
    if i > 0 {
      sums[i] += sums[i-1]
    }
  }

  for k := 2; k <= n; k++ {
    for r := k; r <= n; r++ {
      l := r - k
      dp[l][r] = math.MaxInt
      // Knuth-Yao: optimal split lies in [ip[l][r-1], ip[l+1][r]]
      for i := ip[l][r-1]; i <= ip[l+1][r]; i++ {
        if dp[l][r] > dp[l][i]+dp[i+1][r] {
          dp[l][r] = dp[l][i] + dp[i+1][r]
          ip[l][r] = i
        }
      }
      sum := sums[r-1]
      if l > 0 {
        sum -= sums[l-1]
      }
      dp[l][r] += sum
    }
  }
  fmt.Printf("Minimum cost = %d\n", dp[0][n])

  var buildTree func(l, r int) string
  buildTree = func(l, r int) string {
    if l == r {
      return `"None"`
    }
    if r-l == 1 {
      return fmt.Sprintf(`"%d"`, xs[l])
    }
    i := ip[l][r]
    left := buildTree(l, i)
    right := buildTree(i+1, r)
    return fmt.Sprintf(`("%d", %s, %s)`, xs[i], left, right)
  }

  fmt.Printf("tree.tree(\n %s\n)\n", buildTree(0, n))
}
```