6.4.8 Splitting DP

Splitting DP is a class of techniques where we have some ordered sequence of items and solve a problem by dividing it into segments in some way. We look at three common problems in this space.

6.4.8.1 Segment Sum

Consider the following problem:

> You are given an ordered list of N items. Each range of items from i to j ($i < j$) has an associated value obtained from an arbitrary function, crazy$(i, j + 1)$. Divide the N items into G groups or segments of consecutive items such that the sum of crazy values over the G segments is minimized.

This problem statement leads almost at once to a DP definition:

1. Indices: Left and right endpoints of a range, and amount of groups we want to break that range into.
2. Value: The sum of the crazy values over that range, when broken into the appropriate number of groups.
3. Relation: Iterate through all possible split points between the left and right endpoints and allocation of group count into the two halves, and return the one which results in the minimum sum of the two halves.[75]
4. Iteration Order: The relation depends on smaller ranges and fewer numbers of groups, so we must iterate in increasing range sizes and group counts.
5. Base Case: If the number of groups g is 1, then the the only possibility is to take the crazy value for that range.

Code would appear as follows:

```
int dp[G+1][N+1][N+1]={};
for(int l=0;l<N;l++)
  for(int r=l;r<=N;r++)dp[1][l][r]=crazy(l,r); //base case

for(int g=2;g<=G;g++) //each group count
  for(int l=0;l<N;l++)for(int r=l+g;r<=N;r++){ //each cell
   dp[g][l][r]=INT32_MAX;
   for(int m=l+1;m<r;m++) //each midpoint
     for(int a=1;a<g;a++){ //each assignment of groups to sides
      if(a>m-l||g-a>r-m)continue;
      dp[g][l][r]=min(dp[g][l][r],dp[a][l][m]+dp[g-a][m][r]);
```

[75]If we are breaking a range into, 5 groups, then perhaps 1 group is to the left of the candidate split point, and 4 are to the right. Or 2 to the left and 3 to the right, etc.

```
  }
}

//result is in dp[G][0][N];
```

The DP array is of size $G * N^2$, and each step takes $O(G * N)$ time, leading to an overall runtime of $O(G^2 * N^3)$. This is not great if we expect $G = O(N)$, which would lead to $O(N^5)$. Where is the repeat work? Let's examine the innermost loop where we loop over all combinations of split points and group counts. Suppose the optimal split point is m, with 2 groups in the range $[l, m)$ and 1 in $[m, r)$. In the left range, we must further divide into two groups with ranges $[l, m')$ and $[m', m)$. In this case, our solution includes the elementary intervals $[l, m')$ and $[m', m)$ and $[m, r)$. We find this set of intervals both when we split $[l, r)$ at m with 2 groups on the left and when we split $[l, r)$ at m' with 1 group on the left.

It turns out in our initial DP definition, if we are splitting some range into g groups, we will find the optimial solution $g - 1$ times, once for each of the optimal $g - 1$ split points within that interval which form the g groups. As the order we find the split points in doesn't matter, only their actual location, we can adjust our DP definition and simply look for any of them. For convenience, we will always look for the last split point in the range.

3. Relation: Iterate through all possible **last** split points between the left and right endpoints. As this is the last split point, 1 group lies to the right of it, and the remaining groups lie to the left.

```
int dp[G+1][N+1][N+1]={};
for(int l=0;l<N;l++)
  for(int r=l;r<=N;r++)dp[1][l][r]=crazy(l,r); //base case


for(int g=2;g<=G;g++) //each group count
  for(int l=0;l<N;l++)for(int r=l+g;r<=N;r++){ //each cell
    dp[g][l][r]=INT32_MAX;
    for(int m=l+1;m<r;m++){ //each last split point
      if(g-1>m-l)continue;
      dp[g][l][r]=min(dp[g][l][r],dp[g-1][l][m]+dp[1][m][r]);
    }
}

//result is in dp[G][0][N];
```

With the removal of the most inner loop, we have brought the runtime down to $O(G * N^3)$. In order to improve further, consider the following:

1. Due to the definition of our base case, the term `dp[1][m][r]` resolves exactly to `crazy(m,r)`. After this adjustment, the relation becomes `dp[g][l][r]=min(dp[g][l][r],dp[g-a][l][m]+crazy(m,r))`.

2. After the above adjustment, when computing the entry for any particular l, the only previously computed values we depend on also have a left endpoint of l.

The second point is critical. As noted directly in the code, the ultimate result is stored in `dp[G][0][N]`. If $l = 0$ in this result, and the relation only depends on entries in the DP array with identical values of l, then we need not compute any entries in the DP array for other left hand endpoints($l \neq 0$). This removes an entire dimension from the DP by only considering ranges that start at 0.

1. Indices: The right endpoint r of the range we are considering, and the number of groups g it should be broken into.
2. Value: The sum of the crazy values over the range from $[0, r)$, when broken into the g groups.
3. Relation: Iterate through all possible last split points between the beginning of the array and the right endpoint. Return the one which minimizes the value of `dp[g-1][m]+crazy(m,r)`.
4. Iteration Order: The relation depends on smaller right hand values, and smaller numbers of groups, so we must iterate over increasing right hand endpoints and group counts.
5. Base Cases: `dp[i][1]=crazy(0,i)` as we cannot further breakup a range into fewer than 1 group.

```
int dp[G+1][N+1]={}; //groups and right side of range
for(int r=1;r<=N;r++)dp[1][r]=crazy(0,r); //base cases

for(int g=2;g<=G;g++)for(int r=g;r<=N;r++){ //each group and endpoint
   dp[g][r]=INT32_MAX;
   for(int m=g-1;m<r;m++) //each last split point
      dp[g][r]=min(dp[g][r],dp[g-1][m]+crazy(m,r));
}

//result is in dp[G][N];
```

With the elimination of the left hand endpoint as a dimension, we have reduced the runtime further to $O(G * N^2)$.

6.4.8.2 Divide and Conquer Optimization

Consider the following adjustment to the segment sum problem:

> You are given an ordered list of N items and a value for each item v_i. Each range of items from i to j ($i < j$) has an associated

value which is equal to the sum of the product of the values of each pair of items in the range. This is known as the sum of pairwise products (SoPP).

$$v_{i,j} = \sum_{i \le x < y < j} v_x v_y$$

Divide the N items into G groups of consecutive items such that the sum of the PSoP of each of the G segments is minimized.

This restatement of the problem could be solved with the exact DP arrangement we used early, however if we computed the SoPP each time it was needed, it would add an additional factor of $O(N^2)$. We'll first look at how to eliminate that increase, then look at how we can leverage the structure provided by the SoPP function (relative to crazy) to reduce the overall runtime to less than $O(G * N^2)$.

Fast SoPP Computation Computation of any individual SoPP on the range $[i, j)$ takes $O(N^2)$ time. Naively, computing all such ranges would take an insupportable $O(N^4)$. Instead, if we have computed $v_{i,j}$, can we compute $v_{i,j+1}$ without incurring the complete computation?

$$\begin{aligned}
v_{i,j} + \Delta &= v_{i,j+1} \\
\sum_{i \le x < y < j} (v_x v_y) + \Delta &= \sum_{i \le x < y < j+1} (v_x v_y) \\
\sum_{i \le x < y < j} (v_x v_y) + \Delta &= \sum_{i \le x < y < j} (v_x v_y) + v_j \sum_{i \le x < j} v_x \\
\Delta &= v_{j+1} \sum_{i \le x < j} v_x
\end{aligned}$$

Applying this enables us to incrementally values in linear time for the sum. This can be brought down to constant time by computing the prefix sum of all v_i. With constant time incremental computation the SoPP can be computed for all i, j in $O(N^2)$ time.

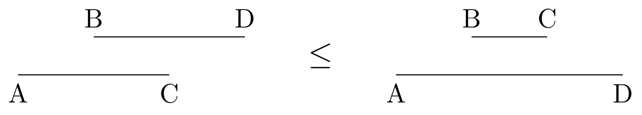
```
int prefix_sum[N+1]={};
for(int i=0;i<N;i++)prefix_sum[i+1]=prefix_sum[i]+v[i];

int sopp[N+1][N+1]={};
for(int i=0;i<N;i++)for(int j=i+1;j<N+1;j++)
   sopp[i][j]=sopp[i][j-1]+v[j-1]*(prefix_sum[j-1]-prefix_sum[i]);
```

Wider is Worse: The Quadrangle Inequality If we examine the SoPP function, we note the value of the function for any large range will be greater than the sum of the values of the sub-ranges which comprise it. More formally,

$v_{i,j} > v_{i,k} + v_{k,j}\ \forall\, i < j < k$. This is a specific case of what is called the *Quadrangle Inequality* or *QI*, known colloquially as "wider is worse".[76] The quadrangle inequality is stated formally as

A function f satisfies the quadrangle inequality if

$$f(a,c) + f(b,d) \leq f(a,d) + f(b,c) \quad \forall\, a \leq b \leq c \leq d$$



The knowledge that wider intervals are disproportionately worse lends some intuition as to how the DP progresses. For a given number of groups g, Consider o_r as the index which minimizes the value of $dp_{g,r}$. From a code perspective, this is the value of `m` which minimizes `dp[g][r]`. How does o_r move as r increases? If wider is worse and we move r to the right, would it make sense for o_r to move to the left? Given the visualization of QI, we know that having two "balanced" segments is better than a longer and a shorter one. It would violate intuition if the optimal split point for $r+1$ lead to less balanced segments than the one for r.

It turns out this intuition is correct. As r increases, so does o_r: $o_r \leq o_{r+1}$. We say that o moves monotonically with the endpoint of the range. A proof that o moves monotonically for all DP instances which share this relation and a cost function which satisfies QI is provided at the end of the section.[77]

Taking Advantage of Monotonicity Consider our naive recursive relation. We iterate over all possible split points and select the one which produces the minimum value. This requires a search of n items for each entry of our DP array. If we take the monotonicity property, we can make an immediate improvement. For `dp[g][r]`, we only need to search splitpoints beginning at o_{r-1} since the optimal split point must be no earlier than that.

```
int dp[G+1][N+1]={}; //groups and right side of range
for(int r=1;r<=N;r++)dp[1][r]=sopp[0][r];

for(int g=2;g<=G;g++){
   int o[N+1]={INT32_MIN}; //optimal split
   for(int r=g;r<=N;r++){
      dp[g][r]=INT32_MAX;
      for(int m=max(g-1,o[r-1]);m<=r-1;m++){
         int t=dp[g-1][m]+sopp[m][r];
         if(t<dp[g][r]){
            dp[g][r]=t;
            o[r]=m;
```

[76] A proof that SoPP satisfies QI is provided at the end of this section.
[77] The cost function in this case is SoPP.

```
            }
        }
    }
}
//answer in dp[G][N]
```

While this saves some time, it does not improve the runtime complexity, only the constant. Consider how we would compute a row of the DP table, `dp[g]` in its entirety. Naively, we compute the values `dp[g][r]` in order, but there are other approaches.

- First compute `dp[g][N/2]` and its split point $o_{N/2}$ by trying all possible split points.
- For all $r > N/2$, $o_r \geq o_{N/2}$
- For all $r < N/2$, $o_r \leq o_{N/2}$
- We have divided the row `dp[g]` into two parts, each with a search space averaging $N/2$ in size.
- We can repeat this process recursively, evaluating the two halves by evaluating the midpoint (with a linear search over the valid split points) and then dividing each in half again.

This is a classic divide-and-conquer problem. We have two problems half the size of the original. Based on master theorem, the runtime for computing that single row must be $O(N * log(N))$ making the overall runtime for computing the entire table, $O(G * N * log(N))$.

Lets look at an example:

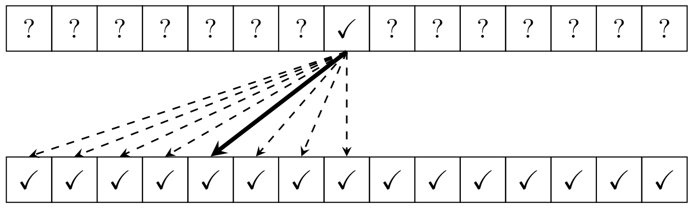


Figure 6.49: When evaluating a new row (with the previous row below complete), we initially evaluate the middle entry, checking every possible split point. We find the optimal solution at entry 4 (0-indexed). We performed $O(N)$ evaluations in this step. Note that split points greater than the entry we are evaluating are not valid, so are skipped. This does not impact the runtime even were we to evaluate them.

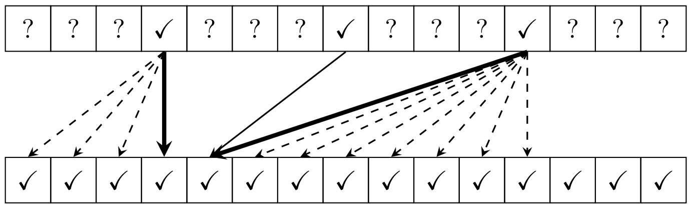


Figure 6.50: We have divided the problem into two subproblems for which we query their respective centers: the 4th entry on the left, and the 12th on the right. Due to the monotonicity property, we know the the optimal split point for the 4th entry must be to the left of the split point for the 8th, and the split point for the 12th entry must be to the right. We find the optimal split points and have done a total of $O(N)$ evaluations for both the 4th and 12th entries combined.

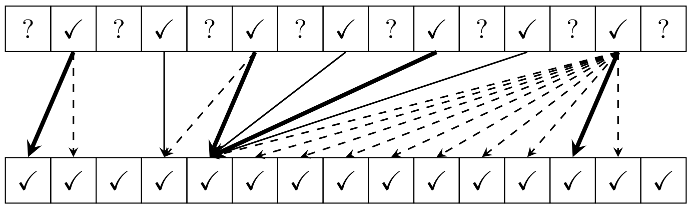


Figure 6.51: We now have 4 subproblems to evaluate the midpoint of, only evaluating entries which are between the optimal split points of the endpoints of a given interval as demanded by the monotonicity principle. We do a total of $O(N)$ evaluations to solve all 4 subproblems.

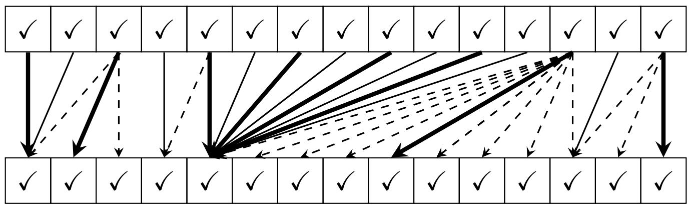


Figure 6.52: We compute the optimal solution for the remaining 8 subproblems, each bounded by the optimal solution at its endpoints. We perform $O(N)$ work total for this stage.

To complete the row, we have performed $O(log(N))$ steps each requiring $O(N)$ work, confirming the $O(N * log(N))$ pre-row runtime.

Implementation The implementation is an adjustment of the middle loop of the unoptimized DP. Instead of looping through all possible endpoints of the range, we do the following:

- Iterate through interval sizes from large to small, dividing by 2 each time. For ease, we assume the intput array is sized at a power of 2, and simply skip any queries which are off the end of the array.[78]
- Iterate through each interval of that size, identifying the left hand side, right hand side, and midpoint (`r`) of the interval. We must take care that all these values are in bounds.
- For each `r`, we identify the lowest and highest possible split points by checking the o value for the LHS and RHS of that range. Due to the order of the computations, we must have already computed these values. Until they are otherwise set, `o[0]` and `o[n]` are set to represent $\pm\infty$ to not improperly bound the search prematurely.[79]

```
int dp[G+1][N+1]={}; //groups and right side of range
for(int i=0;i<N+1;i++)dp[1][i]=sopp[0][i];

for(int g=2;g<=G;g++){
   int o[N+1];//optimal split
   o[0]=-INT32_MAX;o[N]=INT32_MAX;

   //Compute the values of r in the right order, in
   //group sizes of decreasing powers of 2
   int interval_size=1;
   while(interval_size<=N)interval_size<<=1;
   for(;interval_size>1;interval_size>>=1){
      for(int lhs=0;lhs+interval_size/2<=N;lhs+=interval_size){
         int r=lhs+interval_size/2;
         int rhs=min(lhs+interval_size,N);

         //Iterate over all valid split points, bound
         //by the right "o" values
         int temp=-INT32_MAX;
         dp[g][r]=INT32_MAX;
         for(int m=max(g-1,o[lhs]);m<=min(r-1,o[rhs]);m++){
            int t=dp[g-1][m]+sopp[m][r];
            if(t<dp[g][r]){
               dp[g][r]=t;
               temp=m;
            }
```

[78] As we do with binary search and any number of other divide-and-conquer based algorithms.

[79] This usage justifies the `temp` variable in the inner loop. If we were to directly set `o[r]`, then when computing for `r=N`, we would be improperly using `o[r]` as both the upper bound and the result. Similarly we initialize other o values to $-\infty$ to ensure that values which have no valid solution (e.g. $r < g$) do not accidentally artificially limit the search space for later intervals.

```
        }
        o[r]=temp;
      }
    }
}
//answer in dp[k][n]
```

Proofs We prove first that the SoPP function satisfies the quadrangle inequality. We then prove a DP which has a relation as defined by the segment sum problem and a cost function which satisfies QI has an optimal split point which moves monotonically with the endpoints, validating the use of DAC.

Lemma. *Sum of Pairwise Products Satisfies QI.*

Proof. Let f be the sum of pairwise products function. Define

$$\mathrm{comb}(A,B,C) := f(A,C) - \big(f(A,B) + f(B,C)\big)$$

as the difference between f over the individual segments AB and BC, and the combined segment AC, so that

$$f(A,C) = f(A,B) + f(B,C) + \mathrm{comb}(A,B,C)$$

Suppose f does not satisfy QI. Then

$$\exists A,B,C,D : 0 < A \le B \le C \le D$$

such that

$$f(A,C) + f(B,D) > f(A,D) + f(B,C)$$

$$\begin{aligned}
f(A,C) + f(B,D) &> f(A,D) + f(B,C)\\
f(A,C) + f(B,C) + f(C,D) + \mathrm{comb}(B,C,D) &> f(A,D) + f(B,C)\\
f(A,C) + f(C,D) + \mathrm{comb}(B,C,D) &> f(A,D)\\
f(A,C) + f(C,D) + \mathrm{comb}(B,C,D) &> f(A,C) + f(C,D) + \mathrm{comb}(A,C,D)\\
\mathrm{comb}(B,C,D) &> \mathrm{comb}(A,C,D)\\
\mathrm{comb}(B,C,D) &> \mathrm{comb}(A,C,D)
\end{aligned}$$

The right hand side of the final inequality can be broken down for many cost functions. For SoPP, with v_x being the value assigned to a given element:

$$\mathrm{comb}(B,C,D) > \mathrm{comb}(B,C,D) + v_A v_D$$

As there are no values A, D which satisfy the inequality, we have reached a contradiction and therefore f must satisfy QI. □

Corollary. *If a function f satisfies QI, then*

$$\text{comb}(B, C, D) \leq \text{comb}(A, C, D) \quad \forall\, 0 < A \leq B \leq C \leq D \tag{6.1}$$

This is a direct consequence of the final inequality of the proof.

Lemma. *If a DP relation has the form $dp(g, r) = \min_{x<r}(dp(g-1, x)+f(x, r))$, and f satisfies QI, then the optimal value of x moves monotonically with r.*

Proof. Let r be the right hand endpoint of the interval we are examining in our DP for number of groups g, where $g > 1$. Let $dp(g, r)$ be the optimum value of the relation and o_r be the index of the least optimal split point when examining the interval from $[0, r)$. If the optimal split point moves monotonically with the endpoints then

$$o_{r-1} \leq o_r \,\forall\, r$$

Suppose

$$o_r < o_{r-1} \tag{6.2}$$

We begin with the defition of o_{r-1} and decompose it with the comb operator.[80]

$$dp(g-1, o_{r-1}) + f(o_{r-1}, r-1) \quad < \quad dp(g-1, o_r) + f(o_r, r-1)$$

$$\begin{aligned} dp(g-1, o_{r-1}) + f(o_{r-1}, r-1) \quad < \quad & dp(g-1, o_r) + f(o_r, o_{r-1}) \\ & + f(o_{r-1}, r-1) + \text{comb}(o_r, o_{r-1}, r-1) \end{aligned}$$

$$\begin{aligned} & dp(g-1, o_{r-1}) + f(o_{r-1}, r-1) - dp(g-1, o_r) - f(o_r, o_{r-1}) \\ & \qquad - f(o_{r-1}, r-1) - comb(o_r, o_{r-1}, r-1) \quad < \quad 0 \end{aligned}$$

We define the left hand side as k and conclude it is at most 0.

$$\begin{aligned} k \coloneqq\; & dp(g-1, o_{r-1}) + f(o_{r-1}, r-1) - dp(g-1, o_r) - f(o_r, o_{r-1}) \\ & - f(o_{r-1}, r-1) - comb(o_r, o_{r-1}, r-1) \end{aligned} \tag{6.3}$$

$$k < 0 \tag{6.4}$$

We next take a similar definition of o_r.

$$dp(g-1, o_r) + f(o_r, r) \quad \leq \quad dp(g-1, o_{r-1}) + f(o_{r-1}, r)$$

$$\begin{aligned} & dp(g-1, o_r) + f(o_r, o_{r-1}) \\ & + f(o_{r-1}, r-1) + comb(o_r, o_{r-1}, r-1) \\ & \quad + f(r-1, r) + comb(o_r, r-1, r) \quad \leq \quad dp(g-1, o_{r-1}) + f(o_{r-1}, r-1) \\ & \qquad\qquad + f(r-1, r) + comb(o_{r-1}, r-1, r) \end{aligned}$$

[80]Note the definition of o_{r-1} as the least of the optimal split points for $r-1$ means this inequality uses $<$ rather than $\leq$. If the two sides were equal, it would contradict the definition as the least split point.

$$\begin{aligned} comb(o_r, r-1, r) \quad \leq \quad & dp(g-1, o_{r-1}) + f(o_{r-1}, r-1) - dp(g-1, o_r) \\ & - f(o_r, o_{r-1}) - f(o_{r-1}, r-1) \\ & - comb(o_r, o_{r-1}, r-1) + comb(o_{r-1}, r-1, r) \end{aligned} \quad (6.5)$$

We substitute k using equation (6.3) in (6.5).

$$comb(o_r, r-1, r) \leq k + comb(o_{r-1}, r-1, r)$$

$$k \geq comb(o_r, r-1, r) - comb(o_{r-1}, r-1, r) \quad (6.6)$$

Due to (6.1) and (6.2)

$$comb(o_r, r-1, r) \geq comb(o_{r-1}, r-1, r)$$

$$comb(o_r, r-1, r) - comb(o_{r-1}, r-1, r) \geq 0 \quad (6.7)$$

Combining (6.6) and (6.7) yields

$$k \geq comb(o_r, r-1, r) - comb(o_{r-1}, r-1, r) \geq 0$$

$$k \geq 0 \quad (6.8)$$

We have now reached a contradiction as there is no value of k which satisfies (6.4) and (6.8). Therefore we conclude $o_{r-1} \leq o_r$ and the optimal split point moves monotonically with the endpoint. □

Why Can't We Window? A common anti-solution to the segment sum with QI problem is to suggest more optimized linear search. The logic goes:

> We know from QI that the optimal split point for a location $r+1$ must be to the right of the optimal split point for r, so we only need to search from that previous optimal split point. We also know from QI that the function moves monotonically, so we will search from that point until the value of $dp(g-1, m) + f(m, r+1)$ begins to rise, at which point, we know we have found the minimum and can terminate the search.

This windowing approach may be viable for certain relation and would result in linear time to compute a given row of the array. Unfortuantely, it is not correct. The key logical breakdown is QI implies only that the location of the split point moves monotonically with the endpoint of the range, not necessariy that the computation of the relation itself moves unimodally.[81]

The computation of that relation does not, in fact, move unimodally, and we can demonstrate this with a small counter example. Consider an input of $2, 4, 1, 1, 1, 3$, which we intend to split into 3 groups. When computing the final entry of the DP array (3 groups with all 6 numbers), we find the values computed for the relation[82] to be $12, 11, 12, 11$. There are obviously multiple

[81]Has a single minimum or maximum point, as a parabola.

[82]`dp[3][1]+sopp(1,5)` through `dp[3][4]+sopp(4,5)`

local minima, and if the earlier local minimum happened to be greater than the global minimum, we would arrive at the wrong answer.[83]

Given the counter example, solutions must only limit their search based on previously computed split points. They may not rely on the movement of the relation computation itself, and must try every valid split point in the range.

6.4.8.3 Range DP

Consider the following problem:

> You are given an ordered list of N items. Each pair of items i, j ($i < j$) has an associated value obtained from an arbitrary function, match(i, j). Our goal is to match every item in the array with exactly one other item subject to the following constraint: if i and j are matched, any item $x : i < x < j$ may pair only with another such item $y : i < y < j$.[84] Find the pairing with the minimum sum of match(i, j).

To help understand how to break this problem down, consider an input for which we have already computed some pairing. Represent the left hand side of each pair as an open bracket, and the right hand side as a close bracket. The pairing can thus be represented as a series of brackets such as [{}{()()}][{}]. The restriction on how intervals may overlap is equivalent to the pairing being represented by a valid nesting.[85] Any valid nesting of brackets V can be broken down into two recursive rules:

1. (V): A valid nesting surrounded by a pair of matched brackets (of any type)
2. VV: Two consecutive valid nestings

Therfore, if we are trying to find valid nestings, we simply have to check if the above rules apply.

1. Match the first and last character (if possible) and check if the remaining characters constitute a valid nesting.
2. Iterate through all possible split points and check if the two halves are valid nestings.

As these rules will find all possible nestings, if we track the sum of the match value when we apply rule 1, we can determine which matching incurs the lowest cost. Furthermore, each recursive step is executed over a range $[l, r)$ whose result is independent of how brackets match outside of that range. This lends itself to a DP definition: *Range DP*.

[83]Our example has equal local minima, so may arrive at the same final answer, but there are large examples for which this is not the case. This example was chosen merely for its concision. It is, in fact, the smallest example which has an evaluation which is not unimodal.

[84]Put another way, any interval formed by paired points must include either neither or both of the endpoints of any other interval.

[85]as would be used for any mathematical expression

1. Indices: Left and right endpoints of a range.
2. Value: The minimum sum of the match operator for a valid pairing in the range.
3. Relation: $dp(l,r) = \min(\text{match}(l, r-1) + dp(l+1, r-1), \min_{l<x<r}(dp(l,x) + dp(x,r)))$.[86] We either match the endpoints of the range, or iterate through all split points.
4. Iteration Order: The relation depends on smaller ranges, so we iterate in order of increasing range size.
5. Base Case: If the range size is 2, we must match the two elements. If the range size is 1, the pairing is invalid.

The size of the table is N^2 and the amount of work done in each relation is $O(N)$, leading to an overall runtime of $O(N^3)$.

```
int dp[N+1][N+1];
for(int i=0;i<N+1;i++)for(int j=0;j<N+1;j++)dp[i][j]=INT32_MAX;
for(int i=0;i<N-1;i++)dp[i][i+2]=match(i,i+1); //all length 2 intervals

for(int len=4;len<=N;len++)for(int l=0,r=l+len;r<=N;l++,r++){
   if(dp[l+1][r-1]!=INT32_MAX)
      dp[l][r]=min(dp[l][r],match(l,r-1)+(dp[l+1][r-1]));
   for(int m=l;m<=r;m++)if(dp[l][m]!=INT32_MAX&&dp[m][r]!=INT32_MAX)
      dp[l][r]=min(dp[l][r],dp[l][m]+dp[m][r]);
}

//answer in dp[0][N]
```

Relation to Segment Sum From a high level, we seem to do similar things as with the segment sum problem. We iterate over split points and attempt to find minima recursively. Why, then can we not apply the same optimizations to reduce the runtime? There are a couple key differences:

- Segment sum specifies the exact number of groups we break into, but range DP does not. While the exact formulation presented here results in exactly $N/2$ divisions, this is a coincidence rather than a requriemennt.
- With segment sum, once we have determined a split point and ”last” segment, that segment is not further broken down. With range DP, after we have determined a split point, both sides may be further divided. This means the left hand end of the range cannot be eliminated as a dimension in the DP.

[86]The $r-1$ in the match term is due to the fact that we are considering r to be exclusive, but the match function uses the exact index. This differs from the $r-1$ in the dp query, which actually represents a shrinking of the range.

Variations There are many variations on this problem which result in slight changes to the relation:

- Matching identical characters. In this case, we are not concerned with a minimum cost, but only whether it is possible to form a valid matching of only identical characters.[87] This is an adjustment to the first transition, which we can only accept if the characters at the endpoint are equal.

- Allowing mismatches. In this case, instead of always stripping the first and last character in the first transition, we might also strip one or the other, incurring some cost. This might be a goal to minimize the number of of total mismatches, or to minimize the total cost given a fixed maximum number of mismatches. The latter case might introduce a a further dimension to the DP: the number of mismatches we can use.

- Replacement. Many problems in this space involve simply consuming entries in the input and then recursing. Some require instead a replacement of the range with some other value. The overall solution still involves iterating over each possible subrange, and typically the replaced value can be encoded either as the value in the DP array or as an additional dimension.

6.4.8.4 Optimal Breakpoints

6.4.8.5 Knuth's Optimization

[87]AABCCB is valid, but ABABCC is not, as the A and B segments improperly overlap.