MaratonaCIn

Seletiva 2020

Backtracking & Complete Search
Binary Search
Divide and Conquer

Aula 2

Backtracking & Complete Search

Complete Search

MaratonaCIn

- Estratégia geral para resolução de problemas, trata de gerar todas as possíveis soluções e destas encontrar a resposta buscada;
- Ao primeiro momento já vemos que esta solução não é viável para muitos problemas, e de fato ela só é utilizada em subproblemas com espaço de soluções pequeno ou quando o problema original não possui outra solução mesmo;

Mas pra que falar sobre isso?

Complete search tricks

MaratonaCIn

Avaliar todos os subconjuntos de um conjunto de n elementos.

```
1   // recursive
2   int n = 20;
3   vector<int> subset;
4
5   void search(int i) {
6     if(k == n) {
7       get(subset);
8     } else {
9       search(k + 1);
10      subset.push_back(k);
11      search(k + 1);
12      subset.pop_back();
13    }
14  }
15
```

```
1   // iteractive
2   int n = 20;
3   vector<int> subset;
4
5   for(int mask = 0; mask < (1 << n); mask++) {
6     vector<int> subset;
7     for(int j = 0; j < n; j++) {
8       if((mask >> j) & 1) {
9         subset.push_back(j);
10      }
11    }
12    get(subset);
13  }
14
```

Complete search tricks

MaratonaCIn

Para cada subconjunto avaliar todos os subconjuntos de um conjunto de n elementos em O(3^n)

```
1  for(int mask = 0; mask < (1 << n); mask++) {
2      for(int submask = mask; submask > 0; submask = (submask - 1) & mask) {
3          // evaluate here for each submask of mask
4      }
5  }
```

Complete search tricks

MaratonaCIn

Avaliar todos as permutações de um vetor com n elementos em O(n!)

```
1   vector<int> v(n);
2   // fill vector...
3   // v needs to be sorted
4   do {
5   |  get(v);
6   } while(next_permutation(v.begin(), v.end()));
7
```

Meet in the middle

MaratonaCIn

Ideia geral:

- Dividir o espaço de busca em duas (ou mais) partes e fazer busca completa em cada uma delas;
- Juntar as soluções de cada uma das partes para obter a resposta do “problema completo”;

Pontos importantes:

- Esta estratégia é normalmente utilizada quando conseguimos juntar os dois conjuntos de soluções em $O(x)$ ou $O(x*\log(x))$, onde x é o número de soluções;
- Assim conseguimos reduzir a complexidade da busca completa $O(2^n)$ ou $O(n!)$ para algo como $O(n * 2^{(n/2)})$ ou $O((n/2)!)$ que é assintoticamente melhor;

SUMFOUR - 4 values whose sum is 0

MaratonaCIn

Vamos resolver o problema SUMFOUR (link).

Dadas 4 listas de números de tamanho n cada (A, B, C e D), queremos encontrar quantas quadruplas (i, j, k, l) ($0 \le i, j, k, l < n$) existem tal que $A[i] + B[j] + C[k] + D[l] == 0$.

Constraints:

- $0 < n \le 2500$
- $-2^{28} \le A[i], B[i], C[i], D[i] \le 2^{28}$

Solução: encontrar todas as somas para os pares (i, j) e todas as somas para os pares (k, l). Encontra a resposta para o problema contando quantos pares de valores (v, -v) existe tal que v está na primeira solução e -v está na segunda.

SUMFOUR - 4 values whose sum is 0

MaratonaCIn

```
5   const int ms = 2.5e4 + 20;
6
7   int a[ms], b[ms], c[ms], d[ms];
8
9   int main() {
10    int n;
11    scanf("%d", &n);
12    for (int i = 0; i < n; i++) {
13      scanf("%d %d %d %d", a + i, b + i, c + i, d + i);
14    }
15
16    vector<int> first, second;
17    for (int i = 0; i < n; i++) {
18      for (int j = 0; j < n; j++) {
19        first.push_back(-(a[i] + b[j]));
20        second.push_back(c[i] + d[j]);
21      }
22    }
23
24    sort(first.begin(), first.end());
25    sort(second.begin(), second.end());
```

```
27    int l = 0, i = 0;
28    long long ans = 0;
29    while (i < first.size()) {
30      while (l < second.size() && first[i] > second[l]) {
31        l++;
32      }
33      if (first[i] != second[l]) { // second[l] > first[i]
34        i++;
35        continue;
36      }
37
38      int r = l;
39      while (r < second.size() && second[r] == second[l]) {
40        r++;
41      }
42
43      do {
44        ans += r - l;
45        i++;
46      } while (i < first.size() && first[i] == first[i-1]);
47      l = r;
48    }
49    cout << ans << endl;
50  }
```

SUMFOUR - 4 values whose sum is 0

MaratonaCIn

```cpp
using namespace std;
typedef long long ll;

const int ms = 2.5e4 + 20;

int a[ms], b[ms], c[ms], d[ms];

int main() {
  int n;
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d %d %d", a + i, b + i, c + i, d + i);
  }

  vector<int> first, second;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      first.push_back(-(a[i] + b[j]));
      second.push_back(c[i] + d[j]);
    }
  }

  sort(first.begin(), first.end());
  sort(second.begin(), second.end());
  int l = 0, i = 0;
  ll ans = 0;
  while (i < first.size()) {
    int j = i;
    while (j < first.size() && first[i] == first[j]) {
      j++;
    }

    while (l < second.size() && first[i] > second[l]) {
      l++;
    }

    if (first[i] == second[l]) {
      int r = l;
      while (r < second.size() && second[r] == second[l]) {
        r++;
      }

      ans += ll(r - l) * (j - i);
      l = r;
    }

    i = j;
  }
  cout << ans << endl;
}
```

Backtracking - Main idea

MaratonaCIn

- Começamos com uma “solução vazia” e em cada passo buscamos incrementar (ou não exatamente incrementar) a solução atual;
- É importante considerar todas as possíveis modificações em cada passo;
- Normalmente backtracking é levemente melhor que uma busca completa simples pois pode eliminar soluções claramente piores ou impossíveis;
- Otimizações são importantes e podem fazer muita diferença no tempo de execução;

Backtracking - Main idea

MaratonaCIn

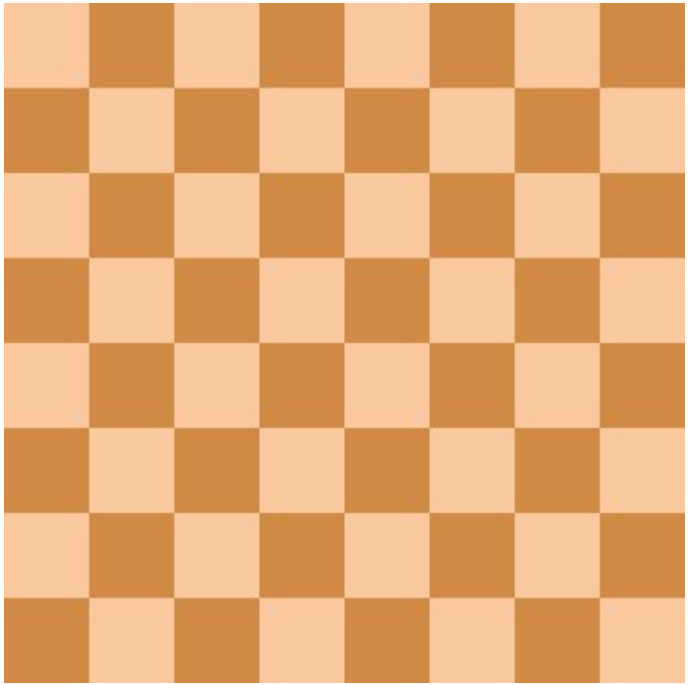
- Estrutura geral para códigos de backtracking;
- Sem as otimizações este código não passa de uma busca completa;
- Normalmente as otimizações são feitas no momento de rejeitar um estado/solução;

```
1   // returns true if found a solution
2   bool backtracking(State state) {
3     if (accept(state)) {
4       output(state);
5       return true;
6     }
7
8     // reject can have optimizations using the data
9     // about previously outputted states
10    if (reject(state)) {
11      return false;
12    }
13
14    bool result = false;
15    for (State nextState : generateStates(state)) {
16      if (backtracking(nextState)) {
17        result = true;
18        // break here if want any solution
19      }
20    }
21    return result;
22  }
```

Backtracking - Well known problems

MaratonaCIn

8 queens problem



Sudoku

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example

MaratonaCIn

Encontrar todos as configurações de jogos em que O ganha no jogo da velha.

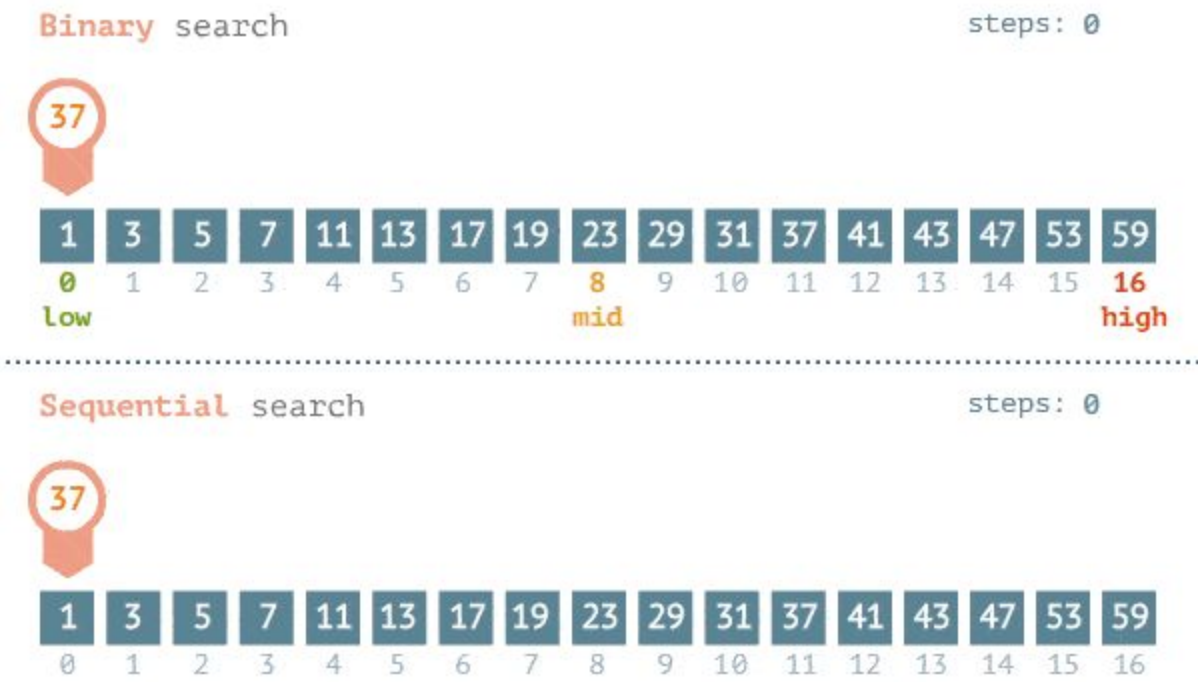
```
36  bool backtracking(vector<string> &board, int turn) {
37      if (isWin(board)) {
38          printSolution(board);
39          return true;
40      }
41
42      if (turn == 9 || isLose(board)) {
43          return false;
44      }
45
46      char curPlayer = (turn & 1) ? 'X' : 'O';
47      bool result = false;
48      for (int i = 0; i < 3; i++) {
49          for (int j = 0; j < 3; j++) {
50              if (board[i][j] != ' ') continue;
51              board[i][j] = curPlayer;
52              result = backtracking(board, turn + 1) || result;
53              board[i][j] = ' ';
54          }
55      }
56      return result;
57  }
```

```
20  bool isWin(vector<string> &board) {
21      return find3(board, 'O');
22  }
23
24  bool isLose(vector<string> &board) {
25      return find3(board, 'X');
26  }
27
28  void printSolution(vector<string> &board) {
29      cout << "========== NEW SOLUTION ==========\n";
30      for (string &x : board) {
31          cout << x << endl;
32      }
33      cout << "==================================\n";
34  }
```

Binary Search

Find elements in a sorted array

MaratonaCIn



http://www.mathwarehouse.com/programming/gifs/binary-vs-linear-search.php

Find elements in a sorted array

MaratonaCIn

- A ideia principal da busca binária é em cada iteração cortar pela metade o espaço de busca;
- Assim achamos a resposta com $\log_2(n)$ passos (*n = v.size()*);

```cpp
bool findElement(const vector<int> &v, int element) {
  int low = 0, high = v.size() - 1;
  while (low < high) {
    int mid = (low + high) / 2;
    if (element <= v[low]) {
      high = mid;
    } else {
      low = mid + 1;
    }
  }
  return v[low] == element;
}
```

STL functions

MaratonaCIn

- lower_bound: primeiro elemento do container que não é menor que um elemento dado;
- upper_bound: primeiro elemento do container que é maior que um elemento dado;
- binary_search: busca por um elemento no container.
- OBS: utilizar métodos implementados da estruturas de dados, quando disponível. Ex: map e set

```
14   vector<string> v;
15   int arr[1000], arr_length = 500;
16   set<pair<int, int>> st;
17
18   lower_bound(v.begin(), v.end(), "lucas"); // returns an iterator
19   upper_bound(arr, arr + arr_length, 100); // returns a pointer
20   st.upper_bound(make_pair(1, 2)); // returns an iterator
```

SUMFOUR - 4 values whose sum is 0

MaratonaCIn

Sabendo das funções, como modificar a solução de SUMFOUR e deixar mais legível?

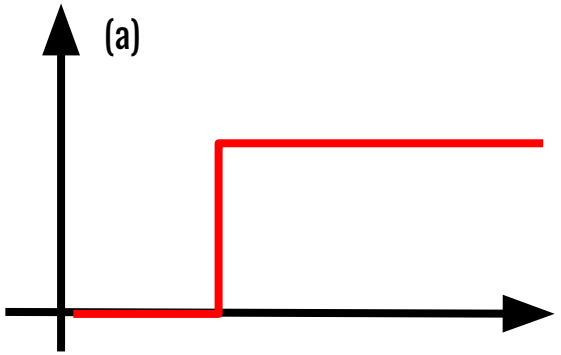
```
5   const int ms = 2.5e4 + 20;
6   int a[ms], b[ms], c[ms], d[ms];
7
8   int main() {
9     int n;
10    cin >> n;
11    for (int i = 0; i < n; i++) {
12      scanf("%d %d %d %d", a + i, b + i, c + i, d + i);
13    }
14
15    vector<int> first, second;
16    for (int i = 0; i < n; i++) {
17      for (int j = 0; j < n; j++) {
18        first.push_back(-(a[i] + b[j]));
19        second.push_back(c[i] + d[j]);
20      }
21    }
```

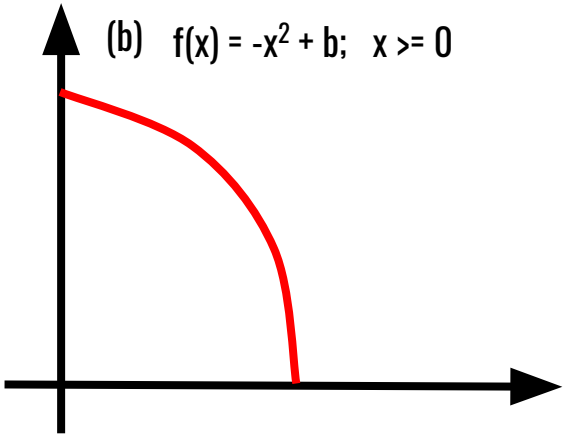
```
22
23    sort(second.begin(), second.end());
24
25    long long ans = 0;
26    for (int v : first) {
27      auto l = lower_bound(second.begin(), second.end(), v);
28      auto r = upper_bound(second.begin(), second.end(), v);
29      ans += r - l;
30    }
31    cout << ans << endl;
32  }
```

Monotonic function

MaratonaCIn

- Uma função é dita monotônica quando é não-decrescente ou não-crescente;
- Podemos encontrar funções que originalmente não são monotônicas mas o trecho que estamos trabalhando é. E.g. o gráfico (b);
- Muitos problemas podem ser abstraídos para uma função monotônica binária, como no gráfico (a);





Binary search on the answer

MaratonaCIn

Ideia geral:

- Buscamos modelar o problema com uma função monotônica (normalmente a binária);
- Fazemos uso desta função para encontrar a resposta por meio de busca binária;

Pontos importantes:

- Normalmente os problemas que são resolvidos com busca binária na resposta buscam o maior/menor elemento que satisfaz alguma propriedade;
- Normalmente a função utilizada não é "*simples*" e é computada fora do código principal da busca;

Fica mais claro com um exemplo :)

Aggressive cows

MaratonaCIn

Vamos resolver o problema problema Aggressive cows (link).

Temos que colocar c vacas em estábulos dado um conjunto n estábulos disponíveis. Os estábulos estão em linha reta nas posições $x[i]$ $(0 \leq i < n)$ dadas. Como as vacas são muito agressivas, queremos fazer isto de maneira que a menor distância entre duas vacas ($|x[k] - x[j]|$) seja máxima.

Qual a maior possível distância mínima entre duas vacas?

Constraints:

- $2 \leq c \leq n \leq 10^5$

Solução: busca binária na resposta (a maior possível distância mínima entre duas vacas).

Aggressive cows

MaratonaCIn

```cpp
const int ms = 1e5 + 20;
int stalls[ms];
int n, c;

int dist(int next_stall, int last_stall) {
  return stalls[next_stall] - stalls[last_stall];
}

bool can(int min_dist) {
  int last_stall = 0;

  for (int i = 1; i < c; i++) {
    int next_stall = last_stall + 1;
    while (next_stall < n &&
           dist(next_stall, last_stall) < min_dist) {
      next_stall++;
    }
    if (next_stall == n) {
      return false;
    }

    last_stall = next_stall;
  }
  return true;
}
```

```cpp
int main() {
  int t;
  cin >> t;
  while (t--) {
    scanf("%d %d", &n, &c);

    for (int i = 0; i < n; ++i) {
      scanf("%d", stalls + i);
    }
    sort(stalls, stalls + n);

    int l = 0, r = 1e9;
    while (l < r) {
      int m = (r + l + 1) / 2;
      if (can(m)) {
        l = m;
      } else {
        r = m - 1;
      }
    }

    printf("%d\n", l);
  }
  return 0;
}
```

Binary search with doubles

MaratonaCIn

- Em alguns casos o espaço de busca da nossa função é “contínuo”;
- Nestes casos as propriedades continuam, mas fazemos iterações até chegar na precisão buscada;
- CUIDADO COM LOOPS INFINITOS;

```
1   double suggestedDoubleBinarySearch(double lo, double hi, int maxIter) {
2     for (int i = 0; i < maxIter; i++) {
3       double mid = (lo + hi) / 2.0;
4       if (binarySearchCheck(mid)) { // generic check here
5         lo = mid;
6       } else {
7         hi = mid;
8       }
9     }
10    return lo;
11  }
```

Divide and Conquer

Main idea

MaratonaCIn

- Dividir o problema original em problemas menores;
- Achar a solução para os problemas menores;
- Utilizar a solução dos problemas menores e algum pré processamento sobre eles para achar a resposta do problema maior;

```
1   T dnq(int left, int right) {
2     if (left == right) {
3       return unitarySolution(left);
4     }
5
6     int mid = (left + right) / 2;
7
8     // calculate the solution for the left and right subproblem
9     T leftSolution = dnq(left, mid);
10    T rightSolution = dnq(mid + 1, right);
11
12    // compose and use left and right solutions to get the answer for the whole segment
13    return composeSolutions(leftSolution, rightSolution, left, right);
14  }
```

Fast Exponentiation

Maratona CIn

Calcular b^e mod m em O(log(e))

```ll
// calculate b ^ e % m
ll fexp(ll b, ll e, ll m) {
  ll res = fexp(b, e / 2, m);
  res = res * res % m;
  if (e&1) {
    res = res * b % m;
  }
  return res;
}
```

Closest pair of points in a set

MaratonaCIn

Dado um conjunto de pontos (x, y), desejamos encontrar o par de pontos com distância mínima;

- Vamos representar os pontos com pares de inteiros;
- Assumimos que a distância ao quadrado entre quaisquer dois pontos pode ser representada por um *long long int*;
- v[] é o vetor com o conjunto de pontos, assumimos que estes pontos estão ordenados por x com desempate por y (função de comparação padrão de std::pair);

```
3   using namespace std;
4
5   typedef long long ll;
6   typedef pair<int, int> ii;
7
8   const int ms = 1e5 + 20;
9   ii v[ms], aux[ms];
10
11  ll sqr(ll x) {
12  |  return x * x;
13  }
14
15  ll dist2(ii a, ii b) {
16  |  ll dx = a.first - b.first;
17  |  ll dy = a.second - b.second;
18  |  return sqr(dx) + sqr(dy);
19  }
```

Closest pair of points in a set

MaratonaCIn

Dado um conjunto de pontos (x, y), desejamos encontrar o par de pontos com distância mínima;

```
21  bool bySecond(ii a, ii b) {
22    return a.second < b.second;
23  }
24
25  ll closestPair(int l, int r) {
26    if (r - l < 2) {
27      return 2e18; // inf
28    }
29
30    int m = (l + r) / 2;
31    ii midPoint = v[m];
32
33    ll d2 = min(closestPair(l, m), closestPair(m, r));
34    // merge two sorted pieces by y
35    copy(v + l, v + r, aux + l);
36    merge(aux + l, aux + m, aux + m, aux + r, v + l, bySecond);
```

```
38    // find points close to midPoint
39    vector<ii> good;
40    for (int i = l; i < r; i++) {
41      if (sqr(v[i].first - midPoint.first) <= d2) {
42        good.push_back(v[i]);
43      }
44    }
45
46    // update closest point distance
47    for (int i = 0; i < good.size(); i++) {
48      // this for runs at most 6 times
49      for (int j = i + 1; j < good.size(); j++) {
50        if (sqr(good[i].second - good[j].second) > d2) {
51          break;
52        }
53        d2 = min(d2, dist2(good[i], good[j]));
54      }
55    }
56
57    return d2;
58  }
```

Closest pair of points in a set

MaratonaCIn

Dado um conjunto de pontos (x, y), desejamos encontrar o par de pontos com distância mínima;

```
21  bool bySecond(ii a, ii b) {
22    return a.second < b.second;
23  }
24
25  // returns the minimum distance between points in [l, r)
26  // and sort(like in mergesort) the interval by y
27  ll closestPair(int l, int r) {
28    if (r - l < 2) {
29      return 2e18; // inf
30    }
31
32    int m = (l + r) / 2;
33    ii midPoint = v[m];
34
35    ll d2 = min(closestPair(l, m), closestPair(m, r));
36    // merge two sorted pieces by y
37    copy(v + l, v + r, aux + l);
38    merge(aux + l, aux + m, aux + m, aux + r, v + l, bySecond);
```

```
40    // find points with x close to midPoint.x
41    vector<ii> good;
42    for (int i = l; i < r; i++) {
43      if (sqr(v[i].first - midPoint.first) <= d2) {
44        good.push_back(v[i]);
45      }
46    }
47
48    // update closest point distance
49    for (int i = 0; i < good.size(); i++) {
50      // for each i there are at most 6 j's to test
51      for (int j = i + 1; j < good.size(); j++) {
52        if (sqr(good[i].second - good[j].second) > d2) {
53          break;
54        }
55        d2 = min(d2, dist2(good[i], good[j]));
56      }
57    }
58
59    return d2;
60  }
```

Maximum subarray sum problem

MaratonaCIn

Encontra o subarray com maior soma em O(n * log(n));
OBS: Este problema pode ser resolvido em O(n) utilizando divide and conquer;

```cpp
int composeSolutions(int leftSolution, int rightSolution, int left, int right) {
  // We computed the best solution contained in the left and right subarray
  // Now we only need to consider solutions starting in the left part and
  // ending in the right
  int mid = (left + right) / 2;

  int maxStartingOnLeft = 0;
  int maxEndingOnRight = 0;

  int auxSum = 0;
  for (int i = mid; i >= l; i--) {
    auxSum += v[i];
    maxStartingOnLeft = max(maxStartingOnLeft, auxSum);
  }

  auxSum = 0;
  for (int i = mid + 1; i <= r; i++) {
    auxSum += v[i];
    maxEndingOnRight = max(maxEndingOnRight, auxSum);
  }

  return max(maxStartingOnLeft + maxEndingOnRight, max(leftSolution, rightSolution));
}
```

```cpp
int maximumSubarraySum(int left, int right) {
  if (left == right) {
    return v[left];
  }

  int mid = (left + right) / 2;

  // calculate the solution for the left and right subproblem
  int leftSolution = maximumSubarraySum(left, mid);
  int rightSolution = maximumSubarraySum(mid + 1, right);

  // compose and use left and right solutions to get the answer for the whole segment
  return composeSolutions(leftSolution, rightSolution, left, right);
}
```