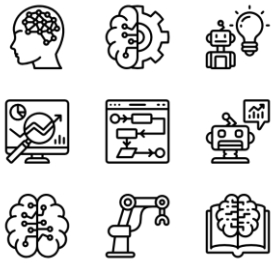
Computer Science for Practicing Engineers

Phương pháp chia để trị



TS. Huỳnh Bá Diệu
Email: dieuhb@gmail.com
Phone: 0914146868

1

Phương pháp Chia để trị (Divide and Conquer)

Các phương pháp thiết kế thuật toán

- Phương pháp Vét cạn (Brute-force or exhaustive search)
- Phương pháp quay lui (Backtracking)
- Phương pháp Chia để trị (Divide and Conquer)
- Phương pháp Qui hoạch động (Dynamic Programming)
- Phương pháp tham lam (Greedy Algorithms)
- Phương pháp nhánh cận (Branch and Bound Algorithm)
- *Phương pháp ngẫu nhiên (Randomized Algorithm)*

2

Phương pháp Chia để trị (Divide and Conquer)

Nội dung:

1. Giới thiệu kỹ thuật chia để trị
2. Thuật toán sắp xếp trộn
3. Thuật toán tính X^n
4. Thuật toán nhân Ấn Độ
5. Thuật toán tìm kiếm Ternary
6. Thuật toán nhân Karatsuba

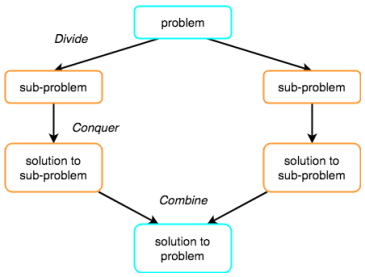
3

Phương pháp chia để trị (divide and conquer)

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

There are mainly three steps in a divide and conquer algorithm

- **Divide** — The problem is broken down or divided into a number of subproblems that are smaller instances of the same problem.
- **Conquer** — The subproblems are solved recursively. If the subproblem sizes are small enough , however, just solve them in a straightforward manner.
- **Combine** — The solution to the subproblems are combined to get the solution for the main problem.



4

Phương pháp chia để trị (divide and conquer)

Advantages

Solving difficult problems

Divide and conquer is a powerful tool for solving conceptually difficult problems: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. Similarly, decrease and conquer only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height n to moving a tower of height n − 1.

Algorithm efficiency

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.

Parallelism

Divide-and-conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

5

Phương pháp chia để trị (divide and conquer)

- **Chia bài toán lớn thành các bài toán con**
- **Giải các bài toán con**
- **Kết hợp lời giải các bài toán con để có lời giải bài toán lớn**

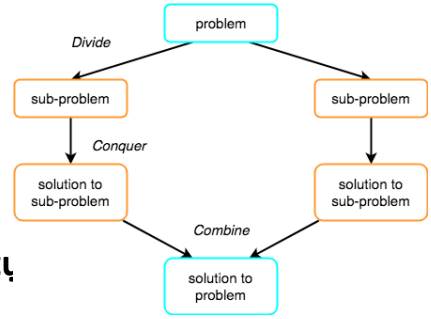
Ví dụ: Sắp xếp dãy a gồm n phần tử

+ Chia thành 2 dãy con L và R

+ Sắp xếp dãy L

+ Sắp xếp dãy R

+ Trộn hai dãy có thứ tự thành 1 dãy có thứ tự



6

Phương pháp chia để trị (divide and conquer)

Merge Sort

A= { 4 1 6 9 2 7 1 3 5 8}

{ 4 1 6 9 2} **{7 1 3 5 8}**

{ 4 1 6 9 2}

{ 4 1} { 6 9 2}

{ 4 1} { 6} { 9 2}

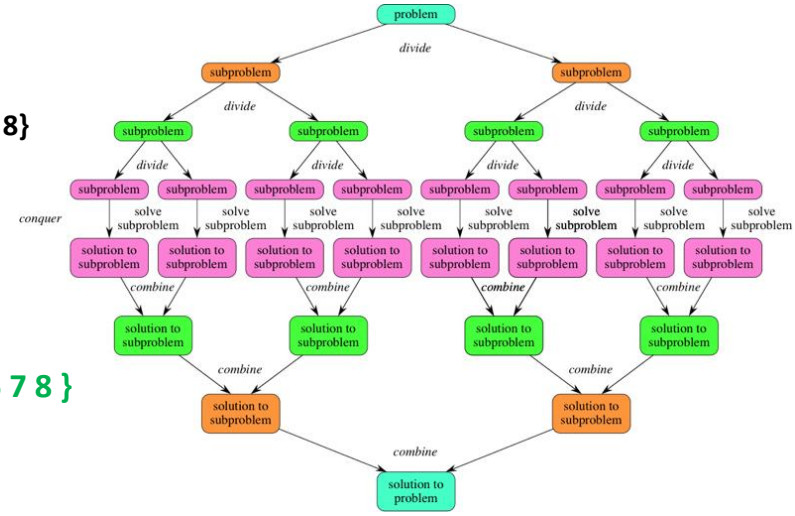
{ 4} {1} { 6} {9} { 2}

{ 1 4 } {6} {2 9}

{ 1 4 } { 2 6 9 }

{1 2 4 6 9 } **{ 1 3 5 7 8 }**

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



7

Phương pháp chia để trị (divide and conquer)

Thuật toán MergeSort sắp xếp dãy từ left (L) đến right (R)

MergeSort(L, R)

If L<R

m= (L+R)/2;

MergeSort(L, m);

MergeSort(m+1, R);

MERGE(L, R);

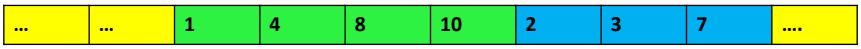
End if

8

Phương pháp chia để trị (divide and conquer)

Thuật toán Merge đoạn từ L đến R (trộn hai dãy có thứ tự thành dãy có thứ tự)

```
Merge(L, R){
 if (L< R){
      + m= (L+R)/2; v=0;
      + i=L; j= m+1;   // i đầu dãy TRAI và j đầu dãy PHAI
      + cấp phát mảng phụ t gồm [R-L +1] phần tử
       + while (i<=m && j<=R)
              nếu a[i]<= a[j] t[v++]= a[i++]; // nếu đầu dãy TRAI nhỏ hơn thì cho vào T
              ngược lại t[v++]= a[j++];       // ngược lại cho đầu dãy PHAi vào T
        + while (i<=m ) t[v++]= a[i++];
        + while (j<=R) t[v++]= a[j++];
        + for (int i=0; i<=R-L; i++) a[L + i] = t[i];
   }
}
```



L=5 R=11

9

Phương pháp chia để trị (divide and conquer)

```
void ms(int ll, int rr)    {
      if(ll<rr)      {
         int m= (ll+rr)/2;
         ms(ll, m);  ms(m+1, rr);
         int i=ll, j=m+1, v=0;
         int []kq= new int[rr-ll+1]; // mang ket qua tam chua day tang dan
          while (i<=m && j<=rr)
                  if(a[i]<=a[j]) { kq[v]=a[i]; i++; v++;}  else { kq[v] =a[j]; v++; j++;}
          while (i<=m) { kq[v]=a[i]; i++; v++;}
          while (j<=rr ) { kq[v] =a[j]; v++; j++;}
          i=ll;  for(v=0; v<kq.length; v++)  a[i++] = kq[v];
      }
}
```

```
void MergeSort()
{
      ms(0, a.length-1);
}
```

10

Phương pháp chia để trị (divide and conquer)

Yêu cầu viết chương trình:

- **Sinh dãy n số nguyên sinh(int n)**
- **In dãy n số inday()**
- **Sắp xếp theo thuật toán MergeSort**

11

Phương pháp chia để trị (divide and conquer)

```
public static void main(String[] args) {
    CSPE_MyP05_Chia_de_tri_MergeSort m= new CSPE_MyP05_Chia_de_tri_MergeSort();
    m.gen(10);
    m.in();
    m.MergeSort();
    m.in();
  }
```

```
Mang A:----------
 5564 2633 5682 6504 6560 258 2800 1259 9205 1382
 ----------------

 Mang A:----------
 258 1259 1382 2633 2800 5564 5682 6504 6560 9205
 ----------------
```

12

Phương pháp chia để trị: Tìm max min

Ví dụ 2: Tìm max min trong dãy A

Cho dãy A. Hãy tìm số max và số min trong dãy.

Cách 1: Duyệt tuần tự, tìm max, tìm min

Cách 2: Tiếp cận theo phương pháp chia để trị

+ Chia làm hai đoạn từ **L→mid** và từ **mid +1 → R**
+ Tìm **max1, min1** trong đoạn L → mid
+ Tìm **max2, min2** trong đoạn mid +1 → R
+ So sánh min1, min2 để tìm **min**, so sánh max1, max2 để tìm **max**

13

Phương pháp chia để trị: Tìm max min

```
void Max_Min()
{
        int Ma =Mi = a[0];
        for(int i=1; i<n; i++)
        {
                if(a[i]>Ma) Ma= a[i];
                if(a[i]<Mi) Mi= a[i];
        }
        SOP("\n Max= " + Ma +" \t Min= " + Mi);
}
void Max_Min(int L, int R) { }
```

14

Phương pháp chia để trị: Tìm max min

```
void Max_Min(int L, int R)   {
   if(L>R);
   else if(L==R) max= min=a[L];
      else      {
         int mid= (L+R)/2;
         int ma1=a[L], mi1=a[L];
         Max_Min(L, mid);
         ma1= max;  mi1=min;
         Max_Min(mid+1,R);
        System.out.println("\n ----------Left = "+ L + "  Right = " +R);
        System.out.println(" max1= " + ma1 + " mi1= " + mi1 +" max2= " + max + " mi2= " + min);
         if(ma1>max) max= ma1;
         if(mi1<min) min= mi1;
      }
   }
```

```
void Tim_MAX_MIN()
   {
      max=a[0];
      min= a[0];
      Max_Min(0, a.length-1);
      System.out.println("\n Max= " + max + "\t Min= " + min);
   }
```

15

Phương pháp chia để trị: Tìm max min

Yêu cầu xây dựng chương trình

- **Sinh dãy số nguyên**
- **Tìm max_min theo phương pháp duyệt tuần tự**
- **Tìm max_min theo phương pháp chia để trị**
- **So sánh thời gian**

```
public static void main(String[] args) {
      CSPE_MyP06_CHIA_DE_TRI_MIN_MAX m= new CSPE_MyP06_CHIA_DE_TRI_MIN_MAX();
      m.gen(10);
      m.in();
      m.Tim_MAX_MIN();
   }
```

16

Phương pháp chia để trị: Tính X^n

Ví dụ: Tính X^n

Thuật toán lặp

```
Kq= 1;
for (int i=1; i<=n; i++ ) Kq= Kq*X;
return Kq;
```

tinh(5,7);

17

Phương pháp chia để trị: Tính X^n

```
public class CSPPE_MyP07_Tinh_X_MU_N {
   long xmun(int x, int n)
   {
      long kq=1;
      for (int i=1; i<=n; i++)          kq=kq*x;
      return kq;
   }
   public static void main(String[] args) {
      CSPPE_MyP07_Tinh_X_MU_N m= new CSPPE_MyP07_Tinh_X_MU_N();
      int n=5;
      System.out.println(" 20^" + n+ "= "+ m.xmun(20, n));
   }
}
```

18

Phương pháp chia để trị: Tính X^n

```
public class CSPPE_MyP07_Tinh_X_MU_N {
   long xmun(int x, int n)
   {
      long kq=1;
      for (int i=1; i<=n; i++)          kq=kq*x;
      return kq;
   }
   public static void main(String[] args) {
      CSPPE_MyP07_Tinh_X_MU_N m= new CSPPE_MyP07_Tinh_X_MU_N();
      int n=50;
      System.out.println(" 20^" + n+ "= "+ m.xmun(20, n));
    }
}
```

Nếu thay n=50, kết quả như thế nào?

19

Phương pháp chia để trị: Tính X^n

Ví dụ: Tính X^n

```
BigInteger X_MU_N(BigInteger x, int n)
   {
      if(n==0) return BigInteger.ONE;
      else return x.multiply(X_MU_N(x, n-1));
   }
```

20

Phương pháp chia để trị: Tính X^n

```
import java.math.BigInteger;
public static void main(String[] args) {
        CSPPE_MyP07_Tinh_X_MU_N m= new CSPPE_MyP07_Tinh_X_MU_N();
        int n=50;
        BigInteger x = new BigInteger("20");
        System.out.println(x+ "^" + n+ "= "+ m.X_MU_N(x, n));
    }
}
```

21

Phương pháp chia để trị: Tính X^n

Phân tích X^n = X*X* X * X* X*…….* X

X^{20} = [X * X * X* X* X* X* X* X* X* X]* [X* X* X* X* X* X* X* X* X* X]

X^{10000} = [X * X * X* X* * * X* X* X* X* X]*[X* X* X* X* X* X* X * ** X* X]

Nhận xét:

Nếu n lẻ thì ta tính $X^n = X * X^{n-1}$

Nếu n chẵn: $X^n = X^{n/2} * X^{n/2}$

Do $X^{n/2}$ giống nhau nên ta chỉ tính 1 lần rồi thực hiện thêm 1 phép nhân, do đó giảm được (n/2) phép tính nhân

Độ phức tạp là logn

Ví dụ n=9 thì cần X* X^4* X^4

$X^4 = X^2 * X^2$; $X^2 = X * X$

Số phép nhân = 1 (khi tính X^2) + 1 (khi tính X^4) + 1 (khi tính X^8) + 1 (khi tính X^9) = 4

22

Phương pháp chia để trị: Tính X^n

Ví dụ: Tính X^n

Phân tích X^n = X*X* X * X* X*…….* X

X^{20} = X * X * X* X* X* X* X* X* X* X* X* X* X* X* X* X* X* X* X* X

X^{10000} = X * X * X* X* * * X* X* X* X* X* X* X* X* X* X* X* X* X * ** X* X

Cần bao nhiêu phép nhân khi tính X^{100} ?

Cần bao nhiêu phép nhân khi tính X^{10000} ?

23

Phương pháp chia để trị: Tính X^n

Ví dụ: Tính X^n

```
BigInteger X_MU_N1(BigInteger x, int n)
  {
    if(n==0) return BigInteger.ONE;
    else
      if(n%2!=0) return x.multiply(X_MU_N1(x, n-1));
      else
      {
        BigInteger kq= X_MU_N1( x, n/2);
        return kq.multiply(kq);
      }
  }
```

24

Phương pháp chia để trị

```
long t1,t2;
t1=System.currentTimeMillis();
      m.ThuatToan();
 t2=System.currentTimeMillis();
System.out.println("\n Thoi gian chay la: " + (t2-t1));
```

```
public static void main(String[] args) {
      CSPPE_MyP07_Tinh_X_MU_N m= new CSPPE_MyP07_Tinh_X_MU_N();
      int n=5200; long t1,t2;
     BigInteger x = new BigInteger("2000");
     BigInteger y= new BigInteger("0");
    t1=System.currentTimeMillis();
      for (int i=1; i<100; i++)
        y=m.X_MU_N1(x, n);
     //System.out.println(x+ "^" + n+ "= "+ m.X_MU_N(x, n));
    t2=System.currentTimeMillis();
   System.out.println("\n Thoi gian chay la: " + (t2-t1));
}
```

BUILD SUCCESSFUL (total time: 9 seconds) BUILD SUCCESSFUL (total time: 2 seconds) *(cách chia để trị)*

25

Phương pháp chia để trị (divide and conquer)

```
public static void main(String[] args) {
      CSPPE_MyP07_Tinh_X_MU_N m= new CSPPE_MyP07_Tinh_X_MU_N();
      int n=9200;
      //System.out.println(" 2^" + n+ "= "+ m.xmun(2, n));
      BigInteger x = new BigInteger("2");
      BigInteger y= new BigInteger("0");
       for (int i=1; i<1000; i++)
         y=m.X_MU_N1(x, n);
      //System.out.println(x+ "^" + n+ "= "+ m.X_MU_N(x, n));
   }
```

Cho n= 9200 và x=2, so sánh thời gian 2 cách tính

26

Phương pháp chia để trị (divide and conquer)

```
public static void main(String[] args) {
       CSPPE_MyP07_Tinh_X_MU_N m= new CSPPE_MyP07_Tinh_X_MU_N();
       int n=9200;
       //System.out.println(" 2^" + n+ "= "+ m.xmun(2, n));
       BigInteger x = new BigInteger("2");
       BigInteger y= new BigInteger("0");
        for (int i=1; i<1000; i++)
           y=m.X_MU_N1(x, n);
       //System.out.println(x+ "^" + n+ "= "+ m.X_MU_N(x, n));
   }
```

Cho n= 9200 và x=2, so sánh thời gian 2 cách tính

Phương pháp chia để trị: Tính X^n

```
X= 4000
X ^ n =
41495155688809929585124078636911611510124462322424368999956573296906528114129081463997070489471
03794288197886611300789182395151075411775307886874834113963687061181803401509523685376000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000Process returned 0 (0x0)   execution time : 2.661 s
Press any key to continue.
```

Tính X^n theo phương pháp lặp

Tính X^n theo pp lặp

```
S=1;
while n>0 {
  nếu n lẻ thì s=s*x;
  n=n/2;
  if(n>0) x=x*x;
}
return S;
```

Thử với 2^{10}

Chạy	S	n	x
0	1	10	2
1	1	5	4
2	1*4	2	16
3	4	1	256
4	1024	0	
5	Dừng vì n=0		

29

Thuật toán nhân Ấn Độ

S=a*b

```
S=0;
while b>0 {
  nếu b lẻ thì S=S+ a;
  b=b/2;
  if(b>0) a=a+a;
}
return S;
```

Thử S=7*23

Chạy	S	a	b
0	0	7	23
1	7	14	11
2			
3			

30

Phương pháp chia để trị: Ternary Search

Ternary search is a divide and conquer algorithm that can be used to find an element in an array. It is similar to binary search where we divide the array into two parts but in this algorithm. In this, we divide the given array into three parts and determine which has the key. We can divide the array into three parts by taking mid1 and mid2 which can be calculated as shown below. Initially, l and r will be equal to 0 and n-1 respectively, where n is the length of the array.

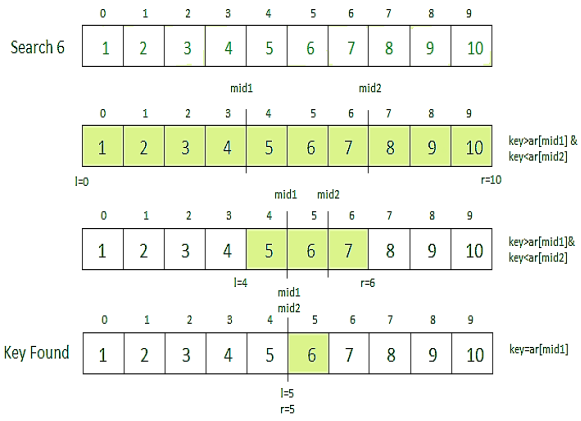
mid1 = l + (r-l)/3
mid2 = r – (r-l)/3

31

Phương pháp chia để trị: Ternary Search

Ternary search

- First, we compare the key with the element at mid1.
- If found equal, we return mid1.
- If not, then we compare the key with the element at mid2. If found equal, we return mid2.
- If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.
- If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.
- If not, then we recur to the second (middle) part.



32

Thuật toán nhân Karatsuba

Anatoly Alexeyevich Karatsuba (31/1/1937 - 28/9/2008)

The Karatsuba algorithm is a fast multiplication algorithm. It was discovered by Anatoly Karatsuba in 1960 and published in 1962.

It reduces the multiplication of two n-digit numbers to at most n^($\log_2 3$). It is therefore faster than the classical algorithm, which n^2 single-digit products.

For example, the Karatsuba algorithm requires $3^{10} = 59{,}049$ single-digit multiplications to multiply two 1024-digit numbers, whereas the classical algorithm requires $(2^{10})^2 = 1{,}048{,}576$ (a speedup of 17.75 times).

33

Thuật toán nhân 2 số lớn có độ dài n

The basic step of Karatsuba's algorithm is a formula that allows one to compute the product of two large x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts.

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as

$$x = x_1 B^m + x_0,$$
$$y = y_1 B^m + y_0,$$

where x_0 and y_0 are less than B^m. The product is then

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0)$$
$$= z_2 B^{2m} + z_1 B^m + z_0,$$

where

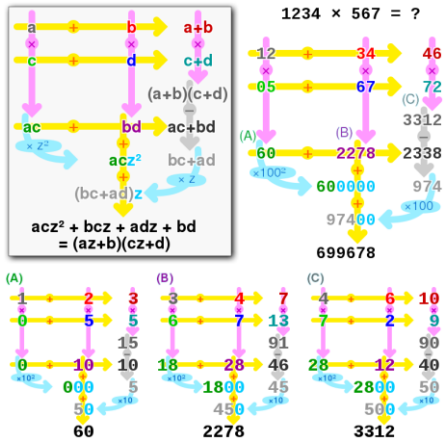
$$z_2 = x_1 y_1,$$
$$z_1 = x_1 y_0 + x_0 y_1,$$
$$z_0 = x_0 y_0.$$

These formulae require four multiplications and were known to Charles Babbage.[4] Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z_0 and z_2 as before one can observe that

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0.$$

An issue that occurs, however, when computing z_1 is that the above computation of $(x_1 + x_0)$ and $(y_1 + y_0)$ may result in overflow (will produce a result in the range $0 \leq \text{result} < 2B^m$), which require a multiplier having one extra bit. This can be avoided by noting that

$$z_1 = (x_0 - x_1)(y_1 - y_0) + z_2 + z_0.$$



34

Thuật toán nhân 2 số lớn có độ dài n

```
procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)        return num1 × num2
    /* Calculates the size of the numbers. */
  m = min(size_base10(num1), size_base10(num2))
  m2 = floor(m / 2)
  /*m2 = ceil(m / 2) will also work */
   /* Split the digit sequences in the middle. */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
   /* 3 calls made to numbers approximately half the size. */
  z0 = karatsuba(low1, low2)
  z1 = karatsuba((low1 + high1), (low2 + high2))
  z2 = karatsuba(high1, high2)
    return (z2 × 10 ^ (m2 × 2)) + ((z1 - z2 - z0) × 10 ^ m2) + z0
```

35

Bài tập về nhà 1

Problem 1. (50 points).

Consider two sums, X = x1 + x2 + . . . + xn and Y = y1 + y2 + . . . + ym.

Give an algorithm that finds indices i and j such that swapping xi with yj makes the two sums equal, that is, X − xi + yj = Y − yj + xi , if they exist.

Analyze your algorithm.

(The amount of credit depends on the correctness of the analysis and the running time of your algorithm.)

36

Bài tập về nhà 2

Problem 2. (50 points).

A game-board has n columns, each consisting of a top number, the cost of visiting the column, and a bottom number, the maximum number of columns you are allowed to jump to the right. The top number can be any positive integer, while the bottom number is either 1, 2, or 3. The objective is to travel from the first column off the board, to the right of the nth column. The cost of a game is the sum of the costs of the visited columns. Assuming the board is represented in a twodimensional array, B[2, n], the following recursive procedure computes the cost of the cheapest game:

```
int CHEAPEST(int i)
  if i > n then return 0 endif;
  x = B[1,i] + CHEAPEST(i+1);
  y = B[1,i] + CHEAPEST(i+2);
  z = B[1,i] + CHEAPEST(i+3);
  case B[2,i] = 1: return x;
       B[2,i] = 2: return min{x,y};
       B[2,i] = 3: return min{x,y,z}
  endcase.
```

(a) Analyze the asymptotic running time of the procedure.
(b) Describe and analyze a more efficient algorithm for finding the cheapest game.

37

Tài liệu đọc thêm

Bernstein, D. J., "Multidigit multiplication for mathematicians"

38

Link YouTube

https://www.youtube.com/watch?v=OtzDFLnIREc

https://www.youtube.com/watch?v=rg517uhXV1o

39