Chapter 5: Divide-and-Conquer

Introduction

Divide-and-conquer is probably the best known general algorithm design technique. Divide-and-conquer algorithms work according to the following general plan:

Step 1. A problem is divided into several subproblems of the same type, ideally of about equal size.

Step 2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).

Step 3. The solutions to the subproblems are combined to get a solution to the original problem.

Example: Finding the maximum value from an array of n numbers (for simplicity, n is a power of 2).

The general divide-and-conquer recurrence

In the most typical case of divide-and-conquer, a problem's instance of size n is divided into $a(> 1)$ instances of size ${}^{n}/_{b}$ where $b > 1$. For simplicity, assuming that size n is a power of b; we get the following recurrence for the running time $T(n)$:

$$T(n) = aT\left({}^{n}/_{b}\right) + f(n)$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size ${}^{n}/_{b}$ and combining their solutions. This recurrence is called the *general divide-and-conquer recurrence*.

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem:

Master theorem: Given the divide-and-conquer recurrence $T(n) = aT\left({}^{n}/_{b}\right) + f(n)$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then:

$$T(n) \in \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta\left(n^{\log_b a}\right) & a > b^d \end{cases}$$

Analogous results hold for the O and Ω notations, too.

Example: Finding the maximum value from an array of n numbers (for simplicity, $n = 2^k$).

```
findMax(a, l, r) {
  if (l == r) return a[l];
  m = (l + r) / 2;
  return max(findMax(a, l, m), findMax(a, m + 1, r));
}
```

The divide-and-conquer recurrence is as follows:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \Theta(1) & n > 1 \\ 0 & n = 1 \end{cases}$$

Example: Finding simultaneously the maximum and minimum values from an array of n numbers.

Algorithm

```
MinMax(l, r, & min, & max) {
  if (l ≥ r - 1)
    if (a[l] < a[r]) {
      min = a[l];
      max = a[r];
    }
    else {
      min = a[r];
      max = a[l];
    }
  else {
    m = ⌊(l + r) / 2⌋;
    MinMax(l, m, minL, maxL);
    MinMax(m + 1, r, minR, maxR);
    min = (minL < minR) ? minL : minR;
    max = (maxL < maxR) ? maxR : maxL;
  }
}
```

The divide-and-conquer recurrence is as follows:

$$C(n) = \begin{cases} C\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + C\left(n - \left\lfloor\frac{n}{2}\right\rfloor\right) + 2 & n > 2 \\ 1 & n \le 2 \end{cases}$$

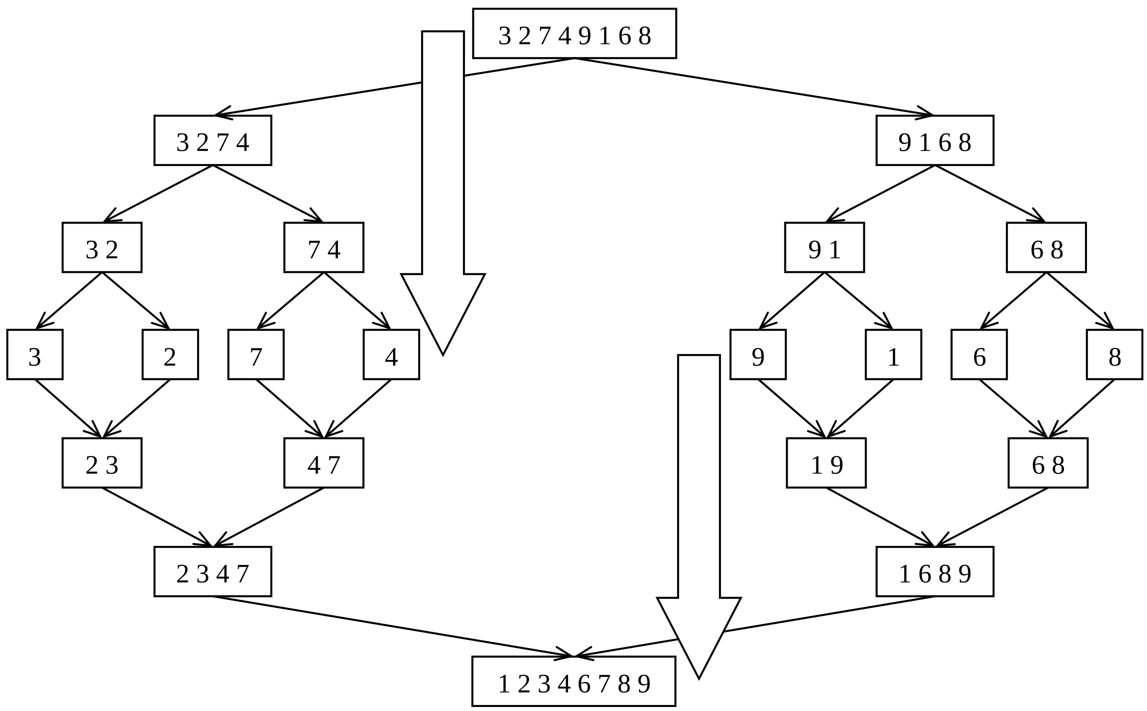
Mergesort

This approach sorts a given array $a_1, a_2, \dots, a_n$ by dividing it into two halves:

$$a_1, a_2, \dots, a_{\lfloor \frac{n}{2} \rfloor}$$

$$a_{\lfloor \frac{n}{2} \rfloor + 1}, a_{\lfloor \frac{n}{2} \rfloor + 2}, \dots, a_n$$

sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.



Algorithm

```
mergeSort(a[1 .. n], low, high) {
  if (low < high) {
    mid = ⌊(low + high) / 2⌋;
    mergeSort(a, low, mid);
    mergeSort(a, mid + 1, high);
    merge(a, low, mid, high);
  }
}
```

```
merge(a[1 .. n], low, mid, high) {
   i = low;
   j = mid + 1;
   k = low;
   while (i ≤ mid) && (j ≤ high)
      if (a[i] ≤ a[j])
         buf[k ++] = a[i ++];
      else
         buf[k ++] = a[j ++];

   if (i > mid)
      buf[k .. high] = a[j .. high];
   else
      buf[k .. high] = a[i .. mid];

   a[low .. high] = buf[low .. high];
}
mergeSort(a, 1, n);
```

How efficient is mergesort?

- Assuming that the key comparison is the basic operation:
  In the best case:

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left\lfloor \frac{n}{2} \right\rfloor & n > 1 \\ 0 & n = 1 \end{cases}$$

Hint: $T(n) \in \Theta(n \log n)$

In the worst case:

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1) & n > 1 \\ 0 & n = 1 \end{cases}$$

Hint: $T(n) \in \Theta(n \log n)$

- Assuming that the assignment statement is the basic operation:

$$M(n) = \begin{cases} M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + M\left(\left\lceil \frac{n}{2} \right\rceil\right) + n & n > 1 \\ 0 & n = 1 \end{cases}$$

Hint: $M(n) \in \Theta(n \log n)$

Quicksort

Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value. This process is called *partition*.

A partition is an arrangement of the array's elements so that all the elements to the left of some element a_s are less than or equal to a_s, and all the elements to the right of a_s are greater than or equal to it:

$$\{a_1 \dots a_{s-1}\} \leq a_s \leq \{a_{s+1} \dots a_n\}$$

After a partition is achieved, a_s will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of a_s independently by the same method.

Algorithm

```
Quicksort(a[left .. right]) {
  if (left < right){
    s = Partition(a[left .. right]);
    Quicksort(a[left .. s - 1]);
    Quicksort(a[s + 1 .. right]);
  }
}
Partition(a[left .. right]) {
  p = a[left];
  i = left;
  j = right + 1;
  do {
    do i++; while (a[i] < p);
    do j--; while (a[j] > p);
    swap(a[i], a[j]);
  } while (i < j);

  swap(a[i], a[j]);
  swap(a[left], a[j]);

  return j;
}
```

Does this design work?

Analysis of Quicksort

For simplicity, assuming that the sequence $a_1, a_2, \ldots, a_n$ contains no duplicate values and the size n is a power of 2: $n = 2^k$. Two comparisons in loops are the basic operation.

In the best case:

$$C_b(n) \in \Theta(n \log n)$$

In the worst case:

$$C_w(n) \in \Theta(n^2)$$

In the average case:

$$C(n) \approx 1.39 n \log_2 n$$

Multiplication of Large Integers

A simple quadratic-time algorithm for multiplying large integers is one that mimics the standard way learned in school. We will develop one that is better than quadratic time.

The basic idea: Observing the multiplication of two complex numbers

$$(a+bi)(c+di) = (ac-bd) + (bc+ad)i$$

K. F. Gauss perceived that:

$$bc + ad = (a+b)(c+d) - (ac+bd)$$

We assume that the data type `large_integer` representing a large integer was constructed. It is not difficult to write linear-time algorithms for three operations: `mul` 10^m, `div` 10^m, and `mod` 10^m.

Let's consider the algorithm that implements the multiplication of two large integers: $u \times v$

Algorithm

```
large_integer MUL(large_integer u, v) {
  large_integer x, y, w, z;

  n = max(number of digits in u, number of digits in v);
  if (u == 0 || v == 0)
    return 0;
  else
    if (n ≤ α)
      return u × v;    // built-in operator
    else {
      m = ⌊n / 2⌋;
      x = u div 10^m;    y = u mod 10^m;
      w = v div 10^m;    z = v mod 10^m;
      return MUL(x, w) mul 10^2m +
             (MUL(x, z) + MUL(y, w)) mul 10^m +
             MUL(y, z);
    }
}
```

Analysis of the algorithm

The divide-and-conquer recurrence is as follows:

$$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + \Theta(n) & n > \alpha \\ 1 & n \le \alpha \end{cases}$$

The Master theorem implies that $T(n) \in \Theta(n^2)$.

Algorithm (upgraded version)

```
large_integer MUL(large_integer u, v, n) {
  n = max(number of digits in u, number of digits in v);

  if (u == 0 || v == 0)
    return 0;
  else
    if (n ≤ α)
      return u × v;
    else {
      m = ⌊n / 2⌋;
      x = u div 10^m;   y = u mod 10^m;
      w = v div 10^m;   z = v mod 10^m;
      r = MUL(x + y, w + z);
      p = MUL(x, w);
      q = MUL(y, z);

      return p mul 10^2m + (r - p - q) mul 10^m + q;
    }
}
```

In this case, the divide-and-conquer recurrence is as follows:

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + \Theta(n) & n > \alpha \\ 1 & n \le \alpha \end{cases}$$

Since $d = 1, a = 3, b = 2$, the Master theorem implies that $T(n) \in \Theta\left(n^{\log_2 3}\right) \approx \Theta(n^{1.585})$.

Extension: Multiplication of two positive integers of n bits. Assuming that n is the power of 2.

Let's x and y be two positive integers of n bits. Obviously:

$$x = 2^{n/2}x_L + x_R$$
$$y = 2^{n/2}y_L + y_R$$

where x_L, x_R are two positive integers represented by $n/2$ leftmost bits and $n/2$ rightmost bits of x, respectively; similarly, y_L, y_R are two positive integers represented by $n/2$ leftmost bits and $n/2$ rightmost bits of y, respectively.

Example: Given $x = 135_{10} = 10000111_2$. Then,

$$x_L = 8_{10}(= 1000_2)$$
$$x_R = 7_{10}(= 0111_2)$$
$$x = 2^{n/2}x_L + x_R = 2^{8/2} \times 8_{10} + 7_{10}$$

Now, we get:

$$x \times y = (2^{n/2}x_L + x_R) \times (2^{n/2}y_L + y_R) = 2^n \times x_L y_L + 2^{n/2} \times (x_L y_R + x_R y_L) + x_R y_R$$

Algorithm

```
int  multiply(x, y) {
  n = max(|x|bit, |y|bit);
  if (n ≤ α)  return   x × y;
  xL = ⌈n / 2⌉ leftmost bits of x;
  xR = ⌊n / 2⌋ rightmost bits of x;
  yL = ⌈n / 2⌉ leftmost bits of y;
  yR = ⌊n / 2⌋ rightmost bits of y;

  r = multiply(xL + xR, yL + yR);
  p = multiply(xL, yL);
  q = multiply(xR, yR);

  return p × 2^n + (r - p - q) × 2^(n/2) + q;
}
```

Strassen's Matrix Multiplication

Suppose we want the product C of two 2×2 matrices, A and B. That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} & a_{11} \times b_{12} + a_{12} \times b_{22} \\ a_{21} \times b_{11} + a_{22} \times b_{21} & a_{21} \times b_{12} + a_{22} \times b_{22} \end{bmatrix}$$

Of course, the time complexity of this straightforward method is $T(n) = n^3$, where n is the number of rows and columns in the matrices. To be specific, the above matrix multiplication requires eight multiplications and four additions.

However, Strassen determined that if we let

$$m_1 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$$
$$m_2 = (a_{21} + a_{22}) \times b_{11}$$
$$m_3 = a_{11} \times (b_{12} - b_{22})$$
$$m_4 = a_{22} \times (b_{21} - b_{11})$$
$$m_5 = (a_{11} + a_{12}) \times b_{22}$$
$$m_6 = (a_{21} - a_{11}) \times (b_{11} + b_{12})$$
$$m_7 = (a_{12} - a_{22}) \times (b_{21} + b_{22})$$

the product C is given by

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Strassen's method requires seven multiplications and 18 additions/subtractions. Thus, we have saved ourselves one multiplication at the expense of doing 14 additional additions or subtractions.

Let A and B be matrices of size $n \times n$, where $n = 2^k$. Let C be the product of A and B. Each of these matrices is divided into four submatrices as follows:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{11} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,\frac{n}{2}} \\ \vdots & \ddots & \vdots \\ c_{\frac{n}{2},1} & \cdots & c_{\frac{n}{2},\frac{n}{2}} \end{bmatrix} \qquad C_{12} = \begin{bmatrix} c_{1,\frac{n}{2}+1} & \cdots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{\frac{n}{2},\frac{n}{2}+1} & \cdots & c_{\frac{n}{2},n} \end{bmatrix}$$
$$C_{21} = \begin{bmatrix} c_{\frac{n}{2}+1,1} & \cdots & c_{\frac{n}{2}+1,\frac{n}{2}} \\ \vdots & \ddots & \vdots \\ c_{n,1} & \cdots & c_{n,\frac{n}{2}} \end{bmatrix} \qquad C_{22} = \begin{bmatrix} c_{\frac{n}{2}+1,\frac{n}{2}+1} & \cdots & c_{\frac{n}{2}+1,n} \\ \vdots & \ddots & \vdots \\ c_{n,\frac{n}{2}+1} & \cdots & c_{n,n} \end{bmatrix}$$

Using Strassen's method, first we compute:

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

where our operations are now matrix addition and multiplication. In the same way, we compute M_2 through M_7. Next we compute

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

and C_{12}, C_{21}, C_{22}. Finally, the product C of A and B is obtained by combining the four submatrices C_{ij}.

Algorithm

```
Strassen(n, A[1..n][1..n], B[1..n][1..n], C[1..n][1..n]) {
  if (n ≤ α)
    C = A × B;
  else {
    "Partition A into 4 submatrices A11, A12, A21, A22";
    "Partition B into 4 submatrices B11, B12, B21, B22";

    Strassen(n/2, A11 + A22, B11 + B22, M1);
    …
    Strassen(n/2, A12 − A22, B21 + B22, M7);

    C11 = M1 + M4 − M5 + M7;
    C12 = M3 + M5;
    C21 = M2 + M4;
    C22 = M1 + M3 − M2 + M6;

    Combine C11, C12, C21, C22 into C;
  }
}
```

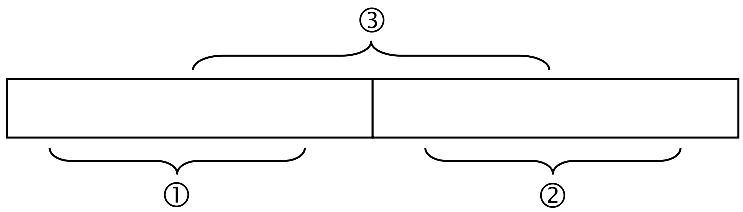
Analysis of the algorithm

The divide-and-conquer recurrence is as follows:

$$T(n) = \begin{cases} 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 & n > \alpha \\ 1 & n \le \alpha \end{cases}$$

The Master theorem implies that: $T(n) \in \Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.81})$

Find the substring with largest sum of elements in an array



Algorithm

```
sumMax(a[1..n], l, r) {
  if (l == r)      return max(a[l], 0);

  c = ⌊(l + r) / 2⌋;
  maxLS = sumMax(a, l, c);
  maxRS = sumMax(a, c + 1, r);

  tmp = maxLpartS = 0;
  for (i = c; i ≥ l; i--) {
    tmp += a[i];
    if (tmp > maxLpartS)   maxLpartS = tmp;
  }
  tmp = maxRpartS = 0;
  for (i = c + 1; i ≤ r; i++) {
    tmp += a[i];
    if (tmp > maxRpartS)   maxRpartS = tmp;
  }
  tmp = maxLpartS + maxRpartS;
  return max(tmp, maxLS, maxRS);
}
max = sumMax(a, 1, n);
```

Analysis of the algorithm

The divide-and-conquer recurrence is as follows:

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n) & n > 1 \\ 0 & n = 1 \end{cases}$$

Hint: $T(n) \in \Theta(n \log n)$

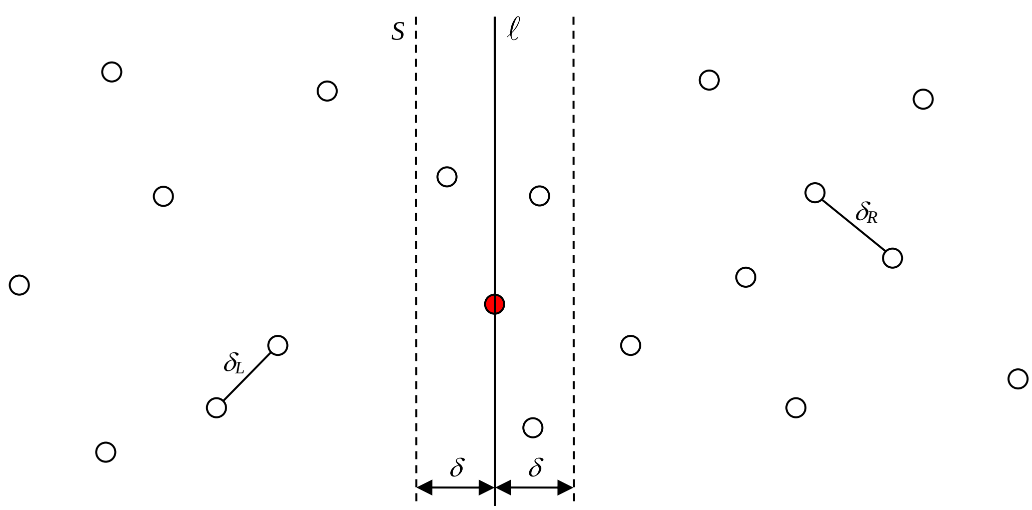
Closest-Pair Problem

Let P be a list of $n > 1$ points in the Cartesian plane: $P = \{p_1, p_2, \dots, p_n\}$. Find a pair of points with the smallest distance between them.

For the sake of simplicity and without loss of generality, we can assume that the points in P are ordered in nondecreasing order of their x coordinate. In addition, let Q be a list of all and only points in P sorted in nondecreasing order of the y coordinate.

If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. Besides, $n = 2{,}3$ is also the stopping condition of the recursive process.

If $n > 3$, we can divide the points into two subsets P_L and P_R of $\left\lceil \frac{n}{2} \right\rceil$ and $\left\lfloor \frac{n}{2} \right\rfloor$ points, respectively, by drawing a vertical line ℓ through the median of their x coordinates so that $\left\lceil \frac{n}{2} \right\rceil$ points lie to the left of or on the line ℓ itself, and $\left\lfloor \frac{n}{2} \right\rfloor$ points lie to the right of or on the line ℓ. Then we can solve the closest-pair problem recursively for subsets P_L and P_R. Let δ_L and δ_R be the smallest distances between pairs of points in P_L and P_R, respectively, and let $\delta = \min\{\delta_L, \delta_R\}$.



Note that δ is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie *on the opposite sides* of the separating line ℓ. Therefore, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width 2δ around the separating line ℓ, since the distance between any other pair of points is at least δ.

Algorithm

```
ClosestPair(Point P[1..n], Point Q[1..n]) {
  if (|P| ≤ 3)
    return the minimal distance found by the brute-force
algorithm;

  ℓ = P[⌈n/2⌉].x;

  Copy the first ⌈n/2⌉ points of P to P_L;
  Copy the same ⌈n/2⌉ points from Q to Q_L;
  Copy the remaining ⌊n/2⌋ points of P to P_R;
  Copy the same ⌊n/2⌋ points from Q to Q_R;

  δ_L = ClosestPair(P_L, Q_L);
  δ_R = ClosestPair(P_R, Q_R);
  δ = min(δ_L, δ_R);

  Copy all the points p of Q for which |p.x - ℓ| < δ into
S[1..k];
  δ_min = δ;
  for (i = 1; i < k; i++) {
    j = i + 1;
    while (j ≤ k) && (|S[i].y − S[j].y| < δ_min) {
      δ_min = min(sqrt((S[i].x − S[j].x)^2 + (S[i].y − S[j].y)^2), δ_min)
      j++;
    }
  }

  return δ_min;
}
```

Analysis of the algorithm

The divide-and-conquer recurrence is as follows:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \in \Theta(n \log n)$$

The change-making problem

Given k denominations: $d_1 < d_2 < \cdots < d_k$ where $d_1 = 1$. Find the minimum number of coins (of certain denominations) that add up to a given amount of money n.

Algorithm

```
moneyChange(d[1..k], money) {
  for (i = 1; i ≤ k; i++)
    if (d[i] == money)
      return 1;

  minCoins = money;
  for (i = 1; i ≤ ⌊money / 2⌋; i++) {
    tmpSum = moneyChange(d, i) + moneyChange(d, money - i);
    if (tmpSum < minCoins)
      minCoins = tmpSum;
  }
  return minCoins;
}
```

Analysis of the algorithm

The divide-and-conquer recurrence is as follows:

$$T(n) = \begin{cases} \sum_{i=1}^{\lfloor n/2 \rfloor} \left(T(i) + T(n-i)\right) + \Theta\left(\frac{n}{2}\right) & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

Hint: $T(n) \in \Omega(2^n)$

Algorithm (upgraded version)

```
moneyChange(d[1..k], money) {
  for (i = 1; i ≤ k; i++)
    if (d[i] == money)
      return 1;

  minCoins = money;
  for (i = 1; i ≤ k; i++)
    if (money > d[i]) {
      tmpSum = 1 + moneyChange(d, money - d[i]);
      if (tmpSum < minCoins)
        minCoins = tmpSum;
    }
  return minCoins;
}
```