2.2 Binary Search & Divide and Conquer (Thuật toán tìm kiếm nhị phân và chia để trị)

Created by TUNG DUC NGUYEN tung2.nguyen, last modified on 2021/05/17

1. Bài toán tìm kiếm

Tìm kiếm là một đòi hỏi rất thường xuyên trong các ứng dụng. Bài toán có thể phát biểu như sau:

Cho một dãy gồm n bản ghi r1,r2,....,r(n). Mỗi bản ghi r(i) (1 ≤ i ≤ n) tương ứng với một khóa k(i). Hãy tìm bản ghi có giá trị khóa bằng X cho trước.

X được gọi là khóa tìm kiếm hay đối trị tìm kiếm (argument).

Công việc tìm kiếm sẽ hoàn thành nếu như có một trong hai tình huống sau xảy ra:

- Tìm được bản ghi có khóa tương ứng bằng X, lúc đó phép tìm kiếm thành công.
- Không tìm được bản ghi nào có khóa bằng X cả, phép tìm kiếm thất bại.

1.1 Tìm kiếm tuần tự (sequential search)

Tìm kiếm tuần tự là một kỹ thuật tìm kiếm đơn giản. Nội dung của nó như sau:

Bắt đầu từ bản ghi đầu tiên, lần lượt so sánh khóa tìm kiếm với khóa tương ứng của các bản ghi trong danh sách, cho tới khi tìm thấy bản ghi mong muốn hoặc đã duyệt hết danh sách.

```
int sequentialSearch(Type k[], Type X) { //tìm kiếm tuần tự trên dãy khóa k1,k2,...,kn. hàm nà
    k[n+1] = X; //Sử dụng khóa phụ để đảm bảo lúc nào cũng tìm thấy kết quả.
    int i = 1;
    while (k[i] != X) ++i; //Sử dụng khóa phụ làm cho vòng lặp không cần kiểm tra điều kiện ->

    if (i == n + 1)
        return 0;
    else
        return i;
}
```

Độ phức tạp của thuật toán tím kiếm tuần tự trong trường hợp tốt nhất là O(1), xấu nhất là O(n) và trung bình là O(n).

1.2 Tìm kiếm nhị phân

Phép tìm kiếm nhị phân có thể áp dụng trên dãy khóa đã có thứ tự: k1 ≤ k2≤ ... ≤ k(n).

Giả sử ta cần tìm trong đoạn k(inf), k(inf +1), ... , k(sup) với khóa tìm kiếm là X, trước hết ta xét khóa nằm giữa dãy k(median) với median = (inf + sup) / 2;

Nếu k(median) < X thì có nghĩa đoạn từ k(inf) tới k(median) chỉ chứa toàn khóa < X, ta tiến hành tìm kiếm tiếp đoạn từ k(median+1) tới k(sup).

Nếu k(median) > X thì có nghĩa đoạn từ k(median) tới k(sup) chỉ chứa toàn khóa > X, ta tiến hành tìm kiếp tới đoạn từ k(inf) tới k(median-1).

Nếu k(median) = X thì việc tìm kiếm thành công (kết thúc quá trình tìm kiếm).

Quá trình tìm kiếm thất bại nếu đến một bước nào đó, đoạn tìm kiếm là rỗng (inf > sup).

```
int binarySearch(Type k[], Type X) {
    int inf = 1, sup = n, median;
    while (inf <= sup) {
        median = (inf + sup) / 2;
        if (k[median] == X)
            return median;
        if (k[median] < X)
            inf = median + 1;
        else
            sup = median -1;
    }
    return 0;
}
```

Độ phức tạp của thuật toán tìm kiếm nhị phân trong trường hợp tốt nhất là O(1), xấu nhất là O(log2(n)) và trung bình cũng là O(log2(n)).

1.3 Cây nhị phân tìm kiếm

Sẽ được giới thiệu sau

2. Thuật toán chia để trị

Chia để trị là một mô hình thiết kế thuật toán dựa trên đệ quy với nhiều phân nhánh.

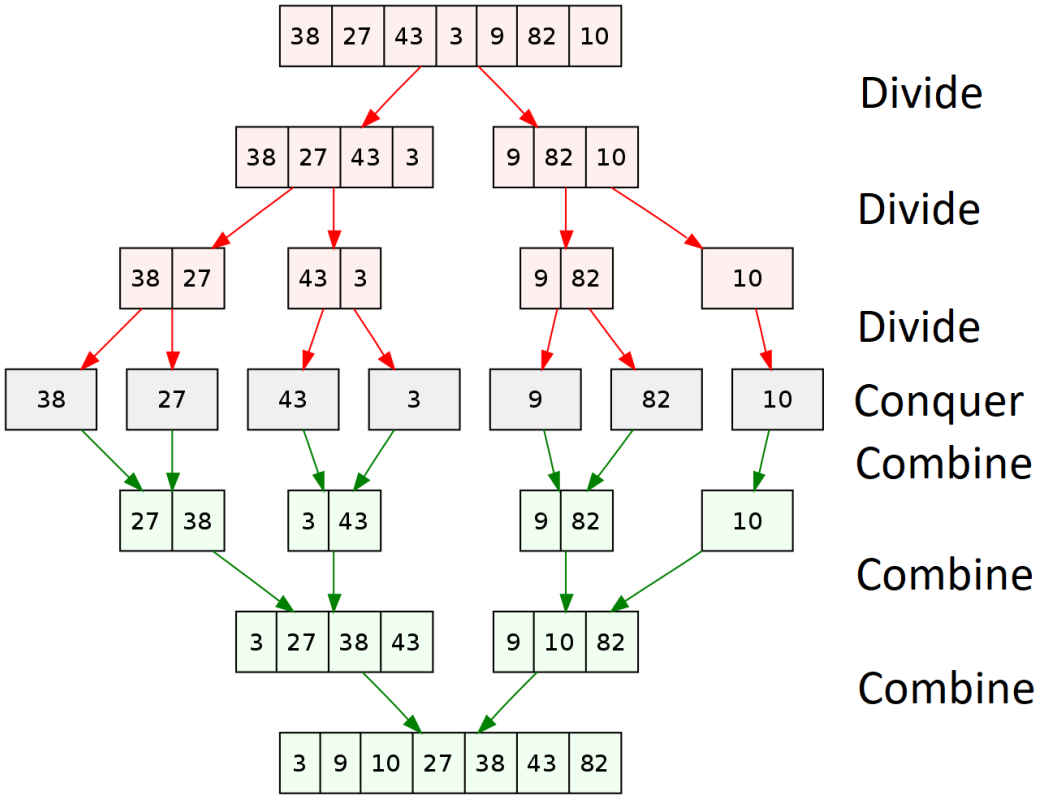
Thuật toán chia để trị (divide and conquer) là chia một cách đệ quy một vấn đề thành hai hoặc nhiều vấn đề con cùng loại hoặc có liên quan, cho đến khi chúng trở nên đủ đơn giản để giải quyết trực tiếp. Các giải pháp của các vấn đề con đó sau đó được kết hợp lại để đưa ra giải pháp cho vấn đề ban đầu.

Kỹ thuật chia để trị là cơ sở của các thuật toán sắp xếp hiệu quả cho nhiều vấn đề, ví dụ: sắp xếp (quick sort, merge sort), nhân các số lớn (thuật toán Karatsuba), tìm cặp điểm gần nhất (finding the closest pair of points), phân tích cú pháp và các tính toán biến đổi Fourier rời rạc.

Việt thiết kế thuật toán chia để trị hiệu quả có thể khó khăn. Tương tự như quy nạp toán học, thông thường cần phải tổng quát hoát vấn đề để biến nó thành giải pháp đệ quy. Tính đúng đắn của thuật toán chia để trị thường được chứng minh bằng quy nạp toán học và chi phí tính toán của nó thường được xác định bằng cách giải các quan hệ lặp đi lặp lại.

Thuật toán chia để trị có thể chia thành ba phần:

1. **Divide** (chia): Chia bài toán thành các bài toán nhỏ hơn. Bước này thường thực hiện một cách đệ quy để chia bài toán nhỏ cho đến khi không chia nhỏ được nữa.
2. **Conquer** (trị): Giải quyết bài toán con đã được chia nhỏ đến mức đơn giản có thể giải quyết trực tiếp.
3. **Combine** (kết hợp): Khi các bài toán nhỏ hơn được giải quyết, kết hợp chúng một cách đệ quy cho đến khi chúng là lời giải cho bài toán ban đầu.



Ví dụ về áp dụng chia để trị cho bài toán sắp xếp

3. Bài toán sắp xếp

Sắp xếp là việc đặt các phần tử của một danh sách theo một thứ tự nhất định. Các thứ tự được sử dụng thường xuyên nhất là thứ tự các số và thứ tự từ vựng. Đầu ra của bài toán sắp xếp phải thoản mã hai điều kiện:

- Đầu ra theo thứ tự không giảm.
- Đâu ra là một hoán vị.

Việc sắp xếp có thể dựa vào so sánh các khóa và đổi chỗ các phần tử cho nhau hoặc không cần sự so sánh (các kỹ thuật đánh dấu).

Một số thuật toán sắp xếp so sánh:

- Sắp xếp nổi bọt (bubble sort)
- Sắp xếp chèn (insertion sort)
- Sắp xếp chọn (selection sort)
- Sắp xếp trộn (merge sort)
- Sắp xếp vun đống (heap sort)
- Sắp xếp nhanh (quick sort)
- ....

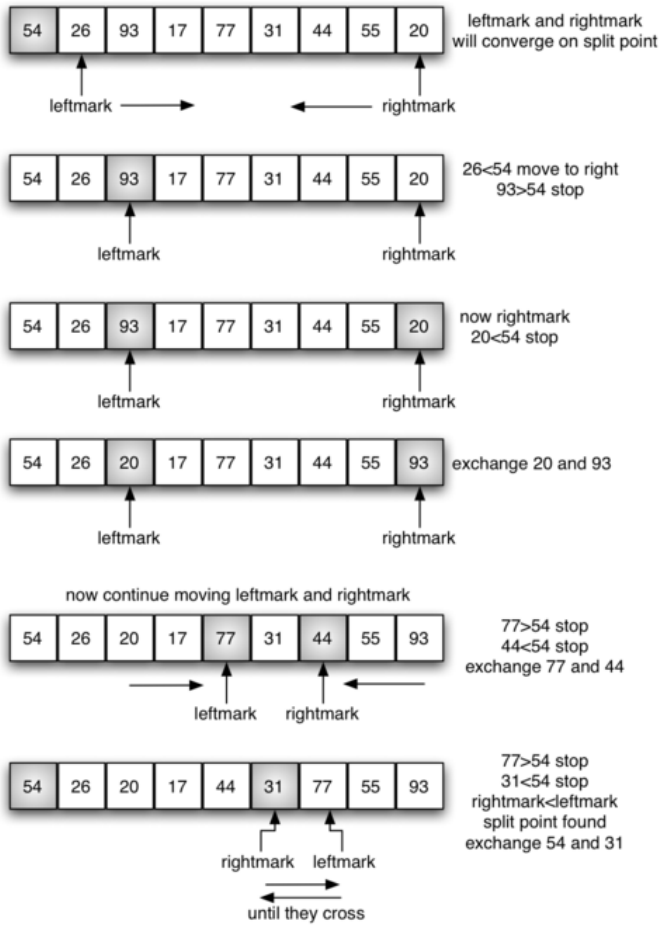
3.1 Thuật toán quick sort

Quick sort là thuật toán sắp xếp tại chỗ - in place - (sắp xếp dựa trên danh sách có sẵn mà không phát sinh thêm bộ nhớ). Nó là một thuật toán thường được sử dụng khi sắp xếp. Khi được triển khai tốt, nó có thể nhanh hơn merge sort và hai hoặc ba lần so với heapsort.

Quick sort là một thuật toán chia để trị. Nó hoạt động bằng cách chọn một phần tử "pivot" từ mảng và chia các phần tử khác thanh hai mảng con, tùy theo việc chúng nhỏ hơn hay lớn hơn pivot. Các mảng con sau đó được sắp xếp một cách đệ quy. Điều này có thể thực hiện tại chỗ (in-place), chỉ yêu cầu một lượng nhỏ bộ nhớ bổ xung để thực hiện việc phân loại. Các bước của thuật toán là:

1. **Kiểm tra kích thước**: nếu số lượng phần tử ít hơn hai (<2), trả về ngay lập tức vì không phải làm gì (dãy được sắp xếp).
2. **Chọn pivot**: chọn một phần tử gọi là pivot trong dãy cần sắp xếp.

3. **Phân vùng**: Sắp xếp lại thứ tự của các phần tử của nó, đồng thời xác định điểm phân chia (vị trí đứng của pivot), sao cho tất cả phần tử nhỏ hơn pivot đứng trước điểm phân chia và tất cả các phần tử lớn hơn nó đứng sau điểm phân chia (các phần tử bằng nó có thể đứng trước hay đứng sau đều được).
4. Gọi đệ quy cho hai dãy con vừa được phân vùng, dãy con từ phần tử đầu điến điểm phân chia, và dãy con từ điểm phân chia đến phần tử cuối.



Thể hiện của thuật toán quick sort như sau:

```
void quicksort(Type list, int lo, int hi) {
    if (low < hi) {
        int p = partition(list, lo, hi);
        quicksort(list, lo, p - 1);
        quicksort(list, p+1, hi);
    }
}

void partition(Type list, int lo, int hi) {
    int pivot = list[lo];
    int left = lo +1, right = hi;
    while (left < right) {
        while (list[left] <= pivot && left <= right) ++left;
        while (list[right] >= pivot && right >= left) --right;
        if (left < right) {
            swap(list[left], list[right]);
        }
    }
```

```
    swap(list[lo], list[right]);
}
```

Việc sắp xếp toàn bộ mảng được thực hiện bằng: ***quicksort(list, 0, list.length() - 1);***

Độ phức tạp của thuật toán quicksort trong best case là O(nlogn), độ phức tạp trung bình O(nlogn) và worst case là O(n^2).

3.2 Thuật toán merge sort

Thuật toán merge sort cũng là một thuật toán thường được sử dụng khi sắp xếp. Hầu hết các triển khai của nó tạo một *stable* sort, có nghĩa là thứ tự các phần tử giống nhau trong cả đầu vào đầu ra.

Merge sort cũng là một thuật toán chia để trị. Nó hoạt động bằng cách chia danh sách chưa được sắp xếp thành n danh sách con, mỗi danh sách chưa một phần tử. Sau đó liên tục hợp nhất cá danh sách con đã được sắp xếp cho đến khi còn lại một danh sách con.

Độ phức tạp của thuật toán merge sort trong best case là O(nlogn), độ phức tạp trung bình O(nlogn) và worst case là O(nlogn).

Các bước mô tả về khái niệm như sau:

1. Chia danh sách chưa được sắp xếp thành n danh sách con, mỗi danh sách chưa 1 phần tử (danh sách một phần tử được coi là đã sắp xếp).
2. Liên tục hợp nhất các danh sách con để tạo ra các danh sách con được sắp xếp mới cho đến khi chỉ còn lại một danh sách con. Đây sẽ là danh sách được sắp xếp.

Có hai cách triển khai thuật toán merge sort, đó là top-down hoặc bottom-up.

Top-down implementation

Do có việc hợp nhất các danh sách con nên ta cần thêm một không gian trung gian để chứa các danh sách con đã được sắp xếp này. Ta tạo một danh sách trung gian đó là B.

```
void topdownMergeSort(Type A[], Type B[], int n) {
    copyArray(A, 0, n, B);
    topdownSplitMerge(B, 0, n, A);  //sort data from B[] to A[]
}

void topdownSplitMerge(Type B[], int iBegin, int iEnd, Type A[]) {
    if (iEnd - iBegin <= 1) return; // nếu size là 1 thì dãy đã được sắp xếp
    int iMiddle = (iEnd + iBegin) / 2; //chia dãy ra làm hai
    topdownSplitMerge(A, iBegin, iMiddle, B);  // xắp xếp dãy bên trái của dãy B, do dó dãy A
    topdownSplitMerge(A, iMiddle, iEnd, B);    // xắp xếp dãy bên phải của dãy B, do dó dãy A
```

```
    topdownMerge(B, iBegin, iMiddle, iEnd, A); //hợp nhất hai dãy đã được sắp xếp nằm trên B t
}

void topdownMerge(Type A[], int iBegin, int iMiddle, int iEnd, Type B[]) {
    int i = iBegin, j = iMiddle;
    for (int k = iBegin; k < iEnd; k++) {
        if (i < iMiddle && (j >= iEnd || A[i] < A[j])) { //nếu dãy bên trái vẫn còn phần tử kế
            B[k] = A[i];
            ++i;
        } else {
            B[k] = A[j];
            ++j;
        }
    }
}

void copyArray(Type A[], int iBegin, int iEnd, Type B[]) {
    for (int i = iBegin; i < iEnd; ++i) {
        B[i] = A[i];
    }
}
```

Việc sắp xếp toàn bộ mảng sẽ được thực hiện bằng: ***topdownMergeSort(A, B, A.length());***

Bottom-up implementation

```
void bottomupMergeSort(Type A[], Type B[], n) {
    for (int width = 1; width < n; width = 2 * width) {
        for (int i = 0; i < n; i = i + 2 * width) {
            bottomupMerge(A, i, min(i+width, n), min(i+2*midth, n), B);
        }
        copyArray(B, 0, n, A);
    }
}

void bottomupMerge(Type A[], int iLeft, int iRight, int iEnd, Type B[]) {
    int i = iLeft, j = iRight;
    for (k = iLeft; k < iEnd; k++) {
        if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            ++i;
        } else {
            B[k] = A[j];
            ++j;
        }
    }
}
```

```
void copyArray(Type A[], int iBegin, int iEnd, Type B[]) {
    for (int i = iBegin; i < iEnd; ++i) {
        B[i] = A[i];
    }
}
```

Việc sắp xếp toàn bộ mảng sẽ được thực hiện bằng: ***bottomupMergeSort(A, B, A.length());***

3.3 Thuật toán heap sort

Sẽ được giới thiệu sau

4. Các bài toán chia để trị

4.1 Tìm tập con liên tục có tổng lớn nhất

https://practice.geeksforgeeks.org/problems/kadanes-algorithm-1587115620/1

Cho một mảng gồm N số nguyên. Tìm mảng con liền nhau có tổng lớn nhất.

Bài toán dùng brute-fore ta có độ phức tạp sẽ là O(n^3), bằng cách xét tổng của tất cả các mảng con liền nhau:

```
int maxSubarraySum(int arr[], int n){
    int maxsum = arr[0];
    for (int i = 0; i < n; i ++) {
        for (int j = i; j < n; j ++) {
            int sum = 0;
            for (int k =i; k <= j; k++) sum+= arr[k];
            if (sum > maxsum)
                maxsum = sum;
        }
    }
    return maxsum;
}
```

Áp dụng chia để trị ta có thể giải bài toán này theo cách sau:

1. Chia mảng ban đầu thành hai mảng
2. Trả ra tổng lớn nhất của:
    - Tổng lớn nhất của mảng bên trái (bằng cách gọi đệ quy)
    - Tổng lớn nhất của mảng bên phải (bằng cách gọi đệ quy)
    - Tổng lớn nhất của mảng gồm phần tử cả phần tử ở giữa

```
int maxCrossingSum(int arr[], int l, int m, int h)
{
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--) {
        sum = sum + arr[i];
        if (sum > left_sum) {
            left_sum = sum;
        }
    }
```

```
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m + 1; i <= h; i++) {
        sum = sum + arr[i];
        if (sum > right_sum) {
            right_sum = sum;
        }
    }
    return max(left_sum + right_sum, left_sum, right_sum);
}
int maxSubArraySum(int arr[], int l, int h)
{
    if (l == h)
        return arr[l];
    int m = (l + h) / 2;
    return max(maxSubArraySum(arr, l, m),
               maxSubArraySum(arr, m + 1, h),
               maxCrossingSum(arr, l, m, h));
}
```

4.2 Tìm cặp điểm gần nhất

https://www.spoj.com/problems/CLOPPAIR/

Cho N điểm trên một mặt phẳng và nhiệm vụ của bạn là tìm một cặp điểm có khoảng cách giữa chúng là nhỏ nhất. Tất cả các điểm sẽ là duy nhất và chỉ có một cặp có khoảng cách nhỏ nhất.

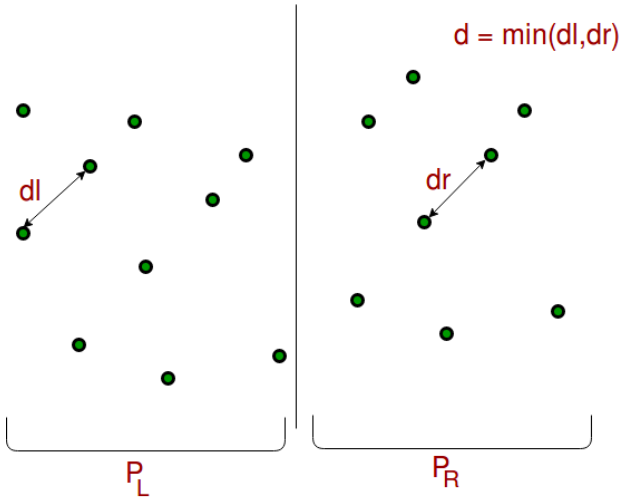
Áp dụng brute-fore: bằng cách tìm khoảng cách của tất cả các cặp điểm để tìm ra đáp án. Độ phức tạp sẽ là O(n^2).

```
struct Point
{
    int x, y;
};

float bruteForce(Point P[], int n) {
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

double dist(Point p1, Point p2) {
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
               );
}
```

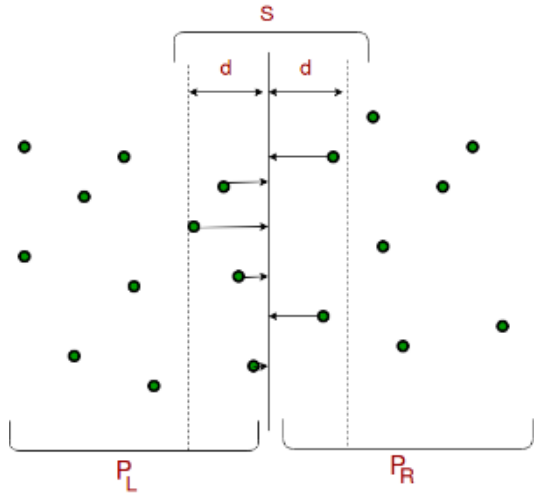
Áp dụng chia để trị, chúng ta có thể giải bài toán này như sau:

1. Sắp xếp tất cả các điểm theo tọa độ X
2. Chia các điểm ra làm hai phần
3. Đệ quy để tìm khoảng cách nhỏ nhất của hai dãy d1 và d2.
4. Lấy khoảng cách nhỏ nhất từ hai dãy con (được gọi là d)
5. Tạo ra một mảng strip[] chứa toàn bộ các điểm mà cách điểm chia (ở bước 2) là <=d.
6. Tìm khoảng cách nhỏ nhất ở trong strip[] (được gọi là k)
7. Trả ra kết quả nhỏ nhất trong hai số k và d.

Step1:



Step5:



Code minh họa:

```
class Point
{
    public:
    int x, y;
};

int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}

int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
           );
}

float bruteForce(Point P[], int n)
```

```
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

float min(float x, float y)
{
    return (x < y)? x : y;
}

float stripClosest(Point strip[], int size, float d)
{
    float min = d; // Initialize the minimum distance as d

    qsort(strip, size, sizeof(Point), compareY);

    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

float closestUtil(Point P[], int n)
{
    if (n <= 3)
        return bruteForce(P, n);

    int mid = n/2;
    Point midPoint = P[mid];

    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n - mid);

    float d = min(dl, dr);

    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    return min(d, stripClosest(strip, j, d) );
}
```

```
float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);

    return closestUtil(P, n);
}
```

5. Discussion questions

1. Lấy ví dụ về worst case và best case của quick sort.
2. Giải thích vì sao độ phức tạp best case của thuật toán quick sort là O(nlogn) và trung bình cũng là O(nlogn).
3. Vì sao độ phức tạp best case của quick sort là O(nlogn) và độ phức tạp của merge sort, heap sort luôn luôn là O(nlogn) mà quicksort lại có thể nhanh hơn?
4. Theo bạn, thuật toán sắp xếp nào được dùng khi sử dụng các hàm sắp xếp của thư viện? Và vì sao?
5. Lấy ví dụ thực tế bạn đã áp dụng chia để trị. Giải thích bạn đã chia, trị và combine như thế nào?

No labels