به نام یگانه معبود بخشنده مهربان

# طراحی الگوریتم ها

# Design and Analysis of Algorithms

گروه مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه اصفهان

ترم دوم سال تحصیلی ۹۰–۹۱

ارائه دهنده : پیمان ادیبی

# روش تقسیم و حل

# Divide-and-Conquer

# تعریف

- روش تقسیم و حل: یک تکنیک طراحی الگوریتم است که با دریافت یک نمونه مسأله که باید حل شود مراحل زیر را انجام می دهد:
  - شکستن آن نمونه مسأله به چند زیرنمونه کوچکتر از همان مسأله (با اندازه ورودی کوچکتر)
  - حل هر زیر نمونه بطور مستقل
  - ترکیب پاسخ زیرنمونه‌ها جهت یافتن پاسخی برای نمونه اصلی
- تقسیم و حل یک روش بالا به پایین (Top-Down) است
- این روش معمولا با روال‌های بازگشتی (Recursive) پیاده سازی می‌شود.

# بخش‌ها

- یک الگوریتم تقسیم و حل معمولاً با چهار مورد زیر مشخص می‌شود:
  - (الف) حد آستانه اندازه ورودی که مسأله به اندازه کوچکتر از آن شکسته نمیشود. ← $n_T$
  - (ب) اندازه زیرنمونه‌ها؛ معمولاً نسبت اندازه نمونه اصلی به زیرنمونه‌ها تعیین می شود ← $k$
  - (پ) تعداد زیرنمونه‌ها ← $r$
  - (ت) الگوریتم ترکیب زیرپاسخ‌ها

گاهی به حد آستانه اندازه ورودی مقدار پایه بازگشتی (recursive base value) نیز گفته میشود.

# شکل عمومی یک الگوریتم تقسیم و حل

```
D_and_C ( int n )
{
   if ( n <= n_T )
         Solve problem without (furthur) subdivision;
   else {
         Split  into r sub-instances each of size n/k;
         For each of r sub-instances
               D_and_C ( n/k );
         Combine the r resulting sub-solutions
         (to produce the solution to the original problem);
   }
}
```

# مثال: جستجوی دودویی (Binary Search) (نسخه بازگشتی)

- مساله: آیا کلید x در آرایهٔ S شامل n کلید وجود دارد؟
- ورودی ها: عدد صحیح مثبت n، آرایهٔ مرتب شده (به ترتیب غیر نزولی) S حاوی کلیدها با اندیس 1 تا n، کلید x
- خروجی ها: اندیس کلید x در آرایه (در صورت وجود) یا 0 (در صورت عدم وجود)

```
index binsearch_ind (index low, index high) {
    if(low>high)
        return 0;
    else{
        index mid = floor((low+high)/2);        // mid = ⌊(low+high)/2⌋;
        if(x==S[mid])
            return mid;
        else if(x<S[mid])
            return binsearch_ind (low, mid-1);
        else
            return binsearch_ind (mid+1, high);
    }
}
```

# مثال: جستجوی دودویی (Binary Search) (نسخه بازگشتی)

- اجرای دستی (trace):

**S=[19,25,28,31,32,35,39,41], n=8, x=45**

```
index binsearch_ind (index low, index high) {
    if(low>high)
        return 0;
    else{
        index mid = floor((low+high)/2);        // mid = ⌊(low+high)/2⌋;
        if(x==S[mid])
            return mid;
        else if(x<S[mid])
            return binsearch_ind (low, mid-1);
        else
            return binsearch_ind (mid+1, high);
    }
}
```

# تحلیل پیچیدگی زمانی جستجوی دودویی (نسخه بازگشتی)

- عمل اصلی: مقایسه x با S[mid] (در زبان اسمبلی دو مقایسه با یک دستور انجام میشوند)
- اندازه ورودی: n
- تحلیل در هر حالت ؟ وجود ندارد چون تعداد تکرار عمل اصلی وابسته به مقادیر ورودیها (x و عناصر S) است.
- تحلیل در بدترین حالت ← W(n)

```
index binsearch_ind (index low, index high) {
    if(low>high)
        return 0;
    else{
        index mid = floor((low+high)/2);
        if(x==S[mid])
            return mid;
        else if(x<S[mid])
            return binsearch_ind (low, mid-1);
        else
            return binsearch_ind (mid+1, high);
    }
}
```

# تحلیل پیچیدگی زمانی جستجوی دودویی (نسخه بازگشتی)

- بدترین حالت: زمانی رخ میدهد که x بزرگتر از همه عناصر S باشد.

  $W(n) = W(n/2)+1$ ← هربار طول آرایه نصف میشود

  $W(1)=1$

  با فرض اینکه n توانی از 2 باشد:

  $W(2)=2, W(4)=3, W(8)=4, W(16)=5, \ldots$

- حدس: $W(n)=1+\lg n$،

  اثبات با استقراء: ...

```
index binsearch_ind (index low, index high) {
    if(low>high)
        return 0;
    else{
        index mid = floor((low+high)/2);
        if(x==S[mid])
            return mid;
        else if(x<S[mid])
            return binsearch_ind (low, mid-1);
        else
            return binsearch_ind (mid+1, high);
    }
}
```

# تحلیل پیچیدگی زمانی جستجوی دودویی (نسخه بازگشتی)

اگر n محدود به توانی از 2 نباشد:

$$W(n) = W(\lfloor n/2 \rfloor) + 1$$

$$W(1) = 1$$

$$\therefore W(n) = \lfloor \lg n \rfloor + 1$$

اثبات با استقراء : ...

(یکبار n را زوج و یکبار فرد بگیرید)

```
index binsearch_ind (index low, index high) {
    if(low>high)
        return 0;
    else{
        index mid = floor((low+high)/2);
        if(x==S[mid])
            return mid;
        else if(x<S[mid])
            return binsearch_ind (low, mid-1);
        else
            return binsearch_ind (mid+1, high);
    }
}
```

- در حالت کلی: $W(n) \in \theta(\lg n)$

اثبات با استفاده از قضیه B.4 : ...

(ابتدا نشان دهید w(n) غیر نزولی نهایی است)

# مثال: مرتب سازی ادغامی (Merge Sort)

- مساله: مرتب کردن n کلید به ترتیب غیر نزولی
- ورودی ها: عدد صحیح مثبت n، آرایهٔ S حاوی کلیدها با اندیسهای 1 تا n
- خروجی ها: آرایهٔ S حاوی کلیدها به ترتیب غیر نزولی

---

ابتدا فرض میکنیم n توانی از 2 باشد

---

- ایده تقسیم و حل:

1. شکستن آرایه به دو زیرآرایه هریک دارای n/2 عنصر
2. مرتب سازی هر زیر آرایه (اگر بقدر کافی کوچک نباشد بصورت بازگشتی عمل میکنیم)
3. ادغام دو زیر آرایه مرتب در یک آرایه مرتب

مثال: [15, 11, 21, 14, 3, 9, 36, 25]

# مرتب سازی ادغامی (Merge Sort)

## Example 2.2

Suppose the array contains these numbers in sequence:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 10 | 12 | 20 | 25 | 13 | 15 | 22 |

1. Divide the array:

   27 10 12 20 and 25 13 15 22

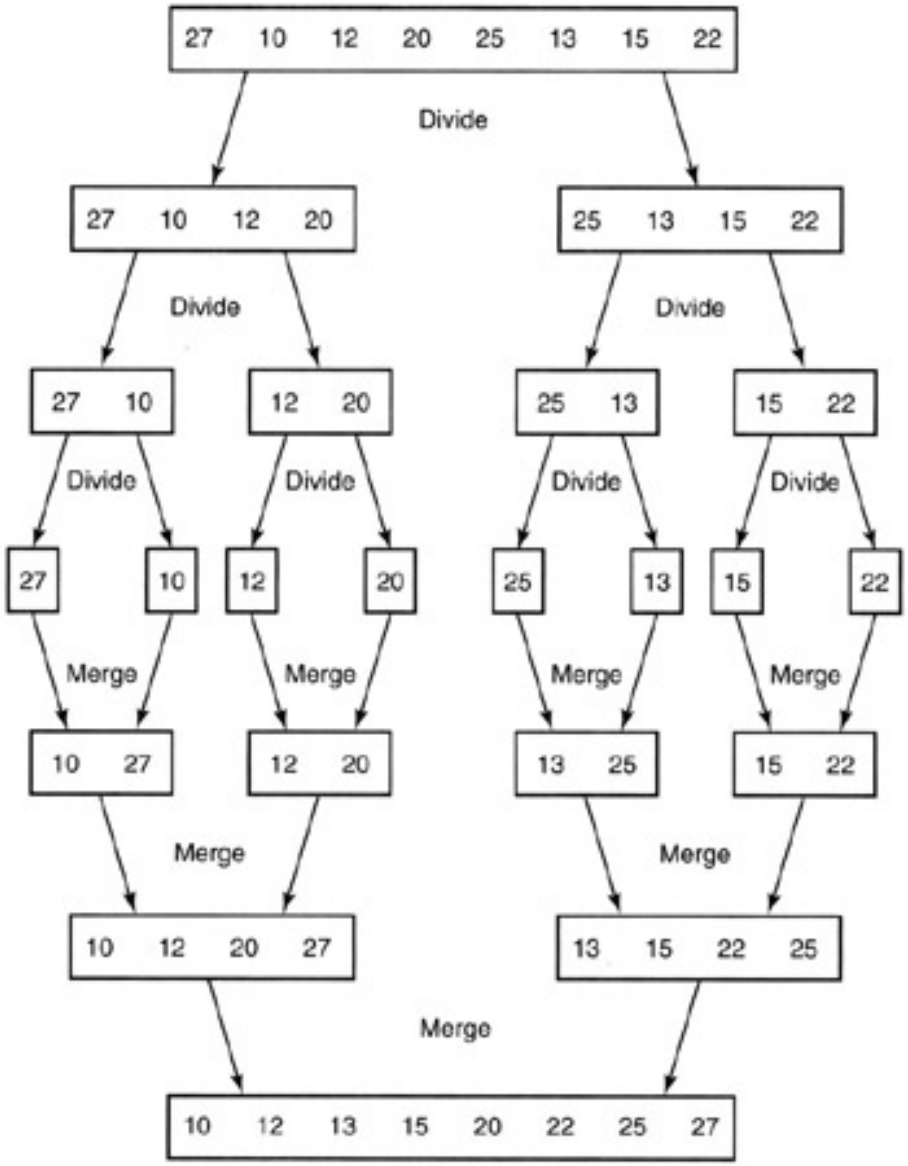
2. Sort each subarray:

   10 12 20 27 and 13 15 22 25

3. Merge the subarrays:

   10 12 13 15 20 22 25 27.

# مرتب سازی ادغامی (Merge Sort)




- یک راه انجام مرحلهٔ 2 بصورت بازگشتی:

  تقسیم را آنقدر ادامه میدهیم تا به یک آرایه تک عنصری برسیم که ذاتاً مرتب است.

# مرتب سازی ادغامی (نسخه اول)

- مساله: مرتب کردن n کلید به ترتیب غیر نزولی
- ورودی ها: عدد صحیح مثبت n، آرایهٔ S حاوی کلیدها با اندیسهای 1 تا n
- خروجی ها: آرایهٔ S حاوی کلیدها به ترتیب غیر نزولی

```
void mergesort (int n, keytype S[])
{
      if (n>1) {
            const int h=[n/2], m = n - h;
            keytype U[1 ..h], V[1 ..m];
            copy S[1] through S[h] to U[1] through U[h];
            copy S[h+1] through S[n] to V[1] through V[m];
            mergesort(h, U);
            mergesort(m, V);
            merge (h, m, U, V, S);
      }
}
```

# الگوریتم ادغام (merge) (نسخه اول)

- مساله: ادغام دو آرایه مرتب در یک آرایه مرتب
- ورودی ها: اعداد h و m، آرایهٔ مرتب U و V بترتیب با اندیسهای 1 تا h و 1 تا m
- خروجی ها: آرایهٔ مرتب S با اندیسهای 1 تا h+m

```
void merge (int h, int m, const keytype U[], const keytype V[], keytype S[]){
  index i, j, k;

  i = 1; j = 1; k = 1;
  while (i <= h && j <= m) {
     if (U[i] < V[j]) {
            S[k] = U[i];
            i++;
       }
       else{
            S[k] = V[j];
            j++;
       }
       k++;
       }
  if (i>h)
        copy V[j] through V[m] to S[k] through S[h+m];
  else
        copy U[i] through U[h] to S[k] through S[h+m];
}
```

- عمل اصلی: مقایسه U[i] و V[j]
- اندازه ورودی: h و m
- تحلیل در هر حالت: ندارد
- بهترین حالت: زمانی که تمام عناصر آرایه با طول کمتر، کوچکتر از عناصر آرایه با طول بیشتر باشد:

  B(h,m)=min(h,m)
- بدترین حالت: زمانی که تمام عناصر بجز آخرین عنصر یکی از آرایه ها، کوچکتر از عناصر آرایه دیگر باشد:

  W(h,m)=h+m-1

# تحلیل مرتب سازی ادغامی (نسخه اول)

- عمل اصلی: مقایسه ای که در تابع ادغام انجام میشود
- اندازه ورودی: n
- تحلیل: در بدترین حالت:

با فرض اینکه n توانی از 2 باشد:

$$W(n) = W(h) + W(m) + (h + m - 1)$$

$$W(n) = 2W(n/2) + n - 1$$

$$W(1) = 0$$

```
void mergesort (int n, keytype S[])
{
      if (n>1) {
            const int h=[n/2], m = n - h;
            keytype U[1 ..h], V[1 ..m];
            copy S[1] through S[h] to U[1] through U[h];
            copy S[h+1] through S[n] to V[1] through V[m];
            mergesort(h, U);
            mergesort(m, V);
            merge (h, m, U, V, S);
      }
}
```

# تحلیل مرتب سازی ادغامی (نسخه اول)

- تحلیل: در بدترین حالت:

$$W(n) = W(h) + W(m) + (h + m - 1)$$

اگر n توانی از 2 باشد:

$$W(n) = 2W(n/2) + n - 1$$

$$W(1) = 0$$

حل با تغییر متغیر و معادله مشخصه ...
(مثال B.19 کتاب)

$$\therefore W(n) = n \lg n - (n - 1)$$

اگر n محدود به توانی از 2 نباشد:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$$

$$W(1) = 0$$

با استفاده از استقرا ثابت میکنیم
W(n) غیر نزولی نهایی است
(مشابه مثال B.25). سپس از قضیه B.4 استفاده میکنیم

$$\therefore W(n) \in \theta(n \lg n)$$

# تحلیل مرتب سازی ادغامی (نسخه اول)

- تحلیل پیچیدگی حافظه :
- تعریف: مرتب سازی درجا (in-place): نیاز به فضای بیش از آرایه اصلی ندارد.
- حافظه مصرفی مرتب سازی ادغامی (نسخه اول):

$$n + n/2 + n/4 + \cdots + 1 = n\sum_{i=0}^{\lg n} 1/2^i = \cdots = 2n - 1$$

```
void mergesort (int n, keytype S[])
{
      if (n>1) {
            const int h=[n/2], m = n - h;
            keytype U[1 ..h], V[1 ..m];
            copy S[1] through S[h] to U[1] through U[h];
            copy S[h+1] through S[n] to V[1] through V[m];
            mergesort(h, U);
            mergesort(m, V);
            merge (h, m, U, V, S);
      }
}
```

- پس مرتب سازی ادغامی یک مرتب سازی درجا نیست
- برای بهبود مصرف حافظه نسخه دوم این الگوریتم ارائه شده است

# مرتب سازی ادغامی (نسخه دوم)

- **مساله**: مرتب کردن n کلید به ترتیب غیر نزولی
- **ورودی ها**: عدد صحیح مثبت n، آرایهٔ S حاوی کلیدها با اندیسهای 1 تا n
- **خروجی ها**: آرایهٔ S حاوی کلیدها به ترتیب غیر نزولی

```
void mergesort2 (index low, index high)
{
    index mid;

    if (low < high) {
        mid = [(low + high)/2];
        mergesort2(low, mid);
        mergesort2(mid + 1, high);
        merge2(low, mid, high);
    }
}
```

- متغیرهای n و S بصورت سراسری تعریف شده‌اند.
- فراخوانی سطح بالا:

  → *mergesort2*(1, *n*);

# الگوریتم ادغام (نسخه دوم)

```
void merge2 (index low, index mid, index high)
{
   index i, j, k;
   keytype U[low .. high];      // A local array needed
                                   // merging
   i = low; j = mid + 1; k = low;
   while (i ≤ mid && j ≤ high){
      if (S[i]< S[j]){
      U[k] = S[i];
      i++;
   }
   else{
      U[k] = S[j];
      j++;
   }
      k++;
}
if (i > mid)
   move S[j] through S[high] to U[k] through U[high];
else
   move S[i] through S[mid] to U[k] through U[high];
move U[low] through U[high] to S[low] through S[high];
}
```

- **مساله:**

  ادغام دو آرایه مرتب در یک آرایه مرتب

- **ورودی ها:**

  سه اندیس low، mid، high، زیرآرایه S[low..high] که بخشهای [low..mid] و [mid+1..high] از قبل بترتیب غیر نزولی مرتب هستند.

- **خروجی ها:**

  زیرآرایهٔ S[low..high] که بترتیب غیر نزولی مرتب شده است.

- **آیا درجاست؟**

  خیر، چون به حافظه اضافی U نیاز دارد

# مرتب سازی ادغامی (نسخه دوم)

- پیچیدگی زمانی در بدترین حالت: مشابه قبل ← $W(n)=\theta(n\lg n)$
- حداکثر حافظه مصرفی: n، که نسبت به نسخه اول تقریبا نصف شده است

```
void mergesort2 (index low, index high)
{
    index mid;

    if (low < high) {
        mid = [(low + high)/2];
        mergesort2(low, mid);
        mergesort2(mid + 1, high);
        merge2(low, mid, high);
    }
}
```

- سؤال: آیا میتوان الگوریتمی بهتر از $\theta(n\lg n)$ برای مرتب سازی پیدا کرد؟
- پاسخ: هر الگوریتم مرتب سازی که از طریق مقایسه کلیدها عمل میکند $\Omega(n\lg n)$ است (اثبات در بخش 7.8 کتاب).

# مثال: ضرب اعداد صحیح بزرگ

- محاسبات بر روی اعداد صحیحی که تعداد ارقام بیش از حد قابل نمایش توسط سخت افزار دارند ← یک راه: استفاده از تکنیک تقسیم و حل
- دو عدد صحیح n رقمی به نام u و v داریم:

$$u = x \times 10^m + y$$

$$v = w \times 10^m + z \quad \rightarrow \quad u \times v = xw \times 10^{2m} + (xz + yw) \times 10^m + yz$$

$$m = \lfloor n/2 \rfloor$$

با چهار ضرب (بصورت بازگشتی) و سه جمع قابل انجام است ← $\theta(n^2)$

- کاهش تعداد ضربها:

$$r = (x + y)(w + z) = xw + (xz + yw) + yz$$

$$xz + yw = r - xw - yz$$

- بدین ترتیب تعداد ضربها از چهار به سه تقلیل می یابد ← $\theta(n^{\lg 3})$
- شکستن بصورت بازگشتی انجام میگیرد تا زمانیکه تعداد ارقام کمتر از حد آستانه قابل نمایش توسط سخت افزار شوند.

# الگوریتم ضرب اعداد صحیح بزرگ

**مساله**: محاسبه حاصلضرب دو عدد صحیح بزرگ u و v

**ورودی ها**: اعداد صحیح بزرگ u و v

**خروجی ها**: prod2 حاصلضرب u و v

```
large_integer prod2 (large_integer u, large_integer v)
{
   large_integer x, y, w, z, r, p, q;
   int n, m;

   n = maximum (number of digits in u, number of digits in v);
   if (u == 0 || v == 0)
      return 0;
   else if (n <= threshold)
      return u × v obtained in the usual way;
   else {
      m = [n/2];
      x = u divide 10^m; y = u rem 10^m;
      w = v divide 10^m; z = v rem 10^m;
      r = prod2(x + y, w + z);
      p = prod2(x, w);
      q = prod2(y, z);
      return p × 10^2m + (r-p-q) × 10^m+q;
   }
}
```

# مثال: ضرب ماتریسها

- مساله: محاسبه حاصلضرب دو ماتریس $n \times n$
- ورودی ها: عدد صحیح مثبت n، دو آرایهٔ دو بعدی A و B با ابعاد $n \times n$
- خروجی ها: آرایهٔ C با ابعاد $n \times n$ که از حاصلضرب $A \times B$ بدست می‌آید
- الگوریتم معمولی (بدون تقسیم و حل) بهبود یافته:

```
void matmult1(int n, n×n_mat A , n×n_mat B , n×n_mat &C){
    index i, j, k;
    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++){
            C[i][j] = A[i][1]*B[1][j];
            for(k=2; k<=n; k++)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
}
```

تعداد ضربها = $n^3$

تعداد جمعها = $n^3 - n^2$

چه عمل اصلی را جمع و چه ضرب بگیریم، مرتبه ← $\theta(n^3)$

# ضرب ماتریسها

- یک ایده تقسیم و حل :

شکستن هر ماتریس به چهار زیرماتریس

$$C = A \times B \quad \rightarrow \quad \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

→ نشان دهید از مرتبهٔ $\theta(n^3)$ است

بهبودی حاصل نشد !

# ضرب ماتریسها به روش استراسن

- یک ایده تقسیم و حل : استراسن نشان داد:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 + M_3 - M_2 + M_6$$

→

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

درستی ایده فوق را بر روی یک مثال بررسی کنید:

# ضرب ماتریسها به روش استراسن

- مساله: محاسبه حاصلضرب دو ماتریس $n \times n$
- ورودی ها: عدد صحیح مثبت n، دو آرایهٔ دو بعدی A و B با ابعاد $n \times n$
- خروجی ها: آرایهٔ C با ابعاد $n \times n$ که از حاصلضرب $A \times B$ بدست می‌آید
- الگوریتم (تقسیم و حل) استراسن:

```
void strassen (int n
                    n × n_matrix A,
                    n × n_matrix B,
                    n × n_matrix& C)
{
   if (n<=threshold)
      compute C = A × B using the standard algorithm;
   else{
      partition A into four submatrices A11, A12, A21, A22;
      partition B into four submatrices B11, B12, B21, B22;
      compute C = A × B using Strassen's method;
         // example recursive call;
         // strassen (n/2, A11 + A22, B11 + B22, M1)
   }
}
```

# ضرب ماتریسها به روش استراسن

- تحلیل پیچیدگی زمانی در هر حالت:
- اندازه ورودی: **n**
- عمل اصلی (حالت اول): یک ضرب عددی (اسکالر)

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 + M_3 - M_2 + M_6$$
$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$T(n) = 7T(n/2)$$
$$T(1) = 1$$

مثال **B.2** کتاب

$$T(n) = n^{\lg 7} = n^{2.81} \in \theta(n^{2.81})$$

# ضرب ماتریسها به روش استراسن

- تحلیل پیچیدگی زمانی در هر حالت:
- اندازه ورودی: **n**
- عمل اصلی (حالت دوم): یک جمع یا تفریق عددی (اسکالر)

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 + M_3 - M_2 + M_6$$
$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$T(n) = 7T(n/2) + 18(n/2)^2$$
$$T(1) = 0$$

مثال **B.20** کتاب

$$T(n) = 6n^{\lg 7} - 6n^2 = 6n^{2.81} - 6n^2 \in \theta(n^{2.81})$$

# ضرب ماتریسها به روش استراسن

- مقایسه با روش معمولی:

| | استراسن | معمولی |
|---|---|---|
| تعداد ضرب ها | $n^{2.81}$ | $n^3$ |
| تعداد جمع ها | $6n^{2.81}-6n^2$ | $n^3-n^2$ |

$n^3$ ⟶ همیشه استراسن بهتر است

$n^3-n^2$ ⟶ برای n های بزرگ استراسن بهتر است

- اگر n محدود به توانی از 2 نباشد:
  - راه اول: اضافه کردن سطر یا ستون صفر تا به توانی از 2 برسد.
  - راه دوم: در فراخوانیهای بازگشتی اگر تعداد سطرها / ستونها فرد شد تنها یک سطر / ستون صفر اضافه کنیم.
  - راه سوم: ...
- حد پایین ضرب ماتریسها: اثبات شده که هر الگوریتم ضرب ماتریسها $\Omega(n^2)$ است. اما هنوز الگوریتمی از مرتبه $\theta(n^2)$ برای این کار ارائه نشده است؛ غیرممکن بودن چنین الگوریتمی نیز به اثبات نرسیده است.

# مثال: مرتب سازی سریع (Quick Sort)

- مساله: مرتب کردن n کلید به ترتیب غیر نزولی
- ورودی ها: عدد صحیح مثبت n، آرایهٔ S حاوی کلیدها با اندیسهای 1 تا n
- خروجی ها: آرایهٔ S حاوی کلیدها به ترتیب غیر نزولی

---

- ایده تقسیم و حل:

1. ابتدا یکی از عناصر آرایه بعنوان عنصر محور (pivot item) در نظر گرفته میشود (معمولاً عنصر اول انتخاب میشود. میتوان یک عنصر را به تصادف انتخاب کرد، یا میانه سه عنصر اول، وسط، و آخر را در نظر گرفت).
2. سپس آرایه به دو بخش شکسته (افراز) میشود، بنحویکه یک بخش حاوی تمام عناصر کوچکتر از عنصر محور و بخش دیگر حاوی تمام عناصر بزرگتر از عنصر محور باشد.
3. سپس هر بخش بطور بازگشتی با همین روش مرتب میشود.

# مرتب سازی سریع

- مثال:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 12 | 16 | 6 | 8 | 23 | 34 | 7 | 45 | 11 |

محور

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 11 | 6 | 8 | 7 | 12 | 34 | 16 | 45 | 23 |

| | | | |
|---|---|---|---|
| 7 | 6 | 8 | 11 |

| | | | |
|---|---|---|---|
| 23 | 16 | 34 | 45 |

| | | |
|---|---|---|
| 6 | 7 | 8 |

| | |
|---|---|
| 16 | 23 |

| |
|---|
| 45 |

| |
|---|
| 6 |

| |
|---|
| 8 |

| |
|---|
| 16 |

# مرتب سازی سریع

- مساله: مرتب کردن n کلید به ترتیب غیر نزولی
- ورودی ها: عدد صحیح مثبت n، آرایهٔ S حاوی کلیدها با اندیسهای 1 تا n
- خروجی ها: آرایهٔ S حاوی کلیدها به ترتیب غیر نزولی
- متغیرهای n و S بصورت سراسری تعریف شده‌اند.
- فراخوانی سطح بالا:

```
quicksort(1, n);
```

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low){
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

# روال افراز (partition)

- مساله: افراز آرایه S برای مرتب سازی سریع
- ورودی ها: دو اندیس low و high، و زیرآرایه S[low..high]
- خروجی ها: اندیس pivotpoint که محل عنصر محور در زیرآرایه را مشخص میکند

```
void partition (index low, index high, index& pivotpoint)
{
    index i, j;
    keytype pivotitem;
    pivotitem = S[low];
    j = low;
    for(i=low+1; i<=high; i++)
        if (S[i] < pivotitem){
                j++;
                exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```

- مثال عددی:

low high

12, 16, 6, 8, 23, 34, 7, 45, 11

.

.

.

# تحلیل پیچیدگی زمانی روال افراز

- عمل اصلی: مقایسه S[i] با pivotitem
- اندازه ورودی: k=high-low+1
- تحلیل در هر حالت: دارد

```
void partition (index low, index high, index& pivotpoint)
{
    index i, j;
    keytype pivotitem;
    pivotitem = S[low];
    j = low;
    for(i=low+1; i<=high; i++)
        if (S[i] < pivotitem){
                j++;
                exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```

T(k)=k-1

# تحلیل پیچیدگی زمانی مرتب سازی سریع

- عمل اصلی: مقایسه S[i] با pivotitem در روال افراز
- اندازه ورودی: n (اندازه آرایه)
- تحلیل در هر حالت: ندارد
- تحلیل بدترین حالت (W(n)) / حالت متوسط (A(n)) / بهترین حالت (B(n))

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low){
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

- بدترین حالت هنگامی است که:
همواره یکی از دو زیرآرایه تهی باشد.
- این وضعیت زمانی رخ میدهد که:
آرایه از ابتدا بصورت غیرنزولی مرتب باشد.
- اثبات مطلب فوق : (صفحه بعد) ←

$$T(n) = \underbrace{T(0)}_{\text{Time to sort left subarray}} + \underbrace{T(n-1)}_{\text{Time to sort right subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

# تحلیل پیچیدگی زمانی مرتب سازی سریع در بدترین حالت

- اگر آرایه بصورت غیرنزولی مرتب باشد:

$$T(n) = \underbrace{T(0)}_{\text{Time to sort left subarray}} + \underbrace{T(n-1)}_{\text{Time to sort right subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

$$\boxed{\begin{aligned} &T(n) = T(n-1) + n - 1 \qquad \text{for } n > 0 \\ &T(0) = 0. \end{aligned}} \longrightarrow T(n) = \frac{n(n-1)}{2}$$

- حال با استقرا ثابت میکنیم برای هر آرایه ورودی زمان اجرا کوچکتر یا مساوی مقدار فوق خواهد بود، یعنی : $W(n) \le \frac{n(n-1)}{2}$
  - پایه استقرا: $W(0) = 0 \le \frac{0(0-1)}{2}$
  - فرض استقرا: برای هر k که k<n داریم: $W(k) \le \frac{k(k-1)}{2}$
  - حکم استقرا: $W(n) \le \frac{n(n-1)}{2}$

# تحلیل پیچیدگی زمانی مرتب سازی سریع در بدترین حالت

- **اثبات حکم استقرا:** فرض کنید p اندیس عنصر محور باشد. داریم:

$$W(n) \le \underbrace{W(p-1)}_{\text{Time to sort left subarray}} + \underbrace{W(n-p)}_{\text{Time to sort right subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

بنابر فرض استقرا (چون p-1<n و n-p<n):

$$W(n) \le \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1$$

$$RHS = p^2 - p + \frac{n(n-1)}{2} + n(1-p)$$

$$= (p-n)(p-1) + \frac{n(n-1)}{2} \le 0 + \frac{n(n-1)}{2}$$

$1 \le p \le n$

$$\boxed{W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)}$$

# تحلیل پیچیدگی زمانی مرتب سازی سریع در بهترین حالت

- بهترین حالت هنگامی است که :

همواره آرایه سطح بالاتر به دو زیرآرایه مساوی افراز شود

- این حالت زمانی رخ میدهد که :

همواره عنصر محور، میانه (median) عناصر آرایه باشد

- با فرض اینکه n توانی از 2 باشد:

$$B(n) = 2B(n/2) + n - 1 \longrightarrow \boxed{B(n) \in \theta(n \lg n)}$$

**a=2, b=2, k=1**

# تحلیل پیچیدگی زمانی مرتب سازی سریع در حالت متوسط

- فرض میکنیم احتمال اینکه اندیس عنصر محور (p) برگردانده شده توسط تابع افراز برابر با هریک از مقادیر 1 تا n باشد، یکسان است. داریم:

$$A(n) = \sum_{p=1}^{n} \rho_p \alpha_p = \sum_{p=1}^{n} \frac{1}{n}\Big[\underbrace{(n-1)}_{\text{Time to partition}} + \underbrace{A(p-1)}_{\text{Avg. Time to sort left subarray}} + \underbrace{A(n-p)}_{\text{Avg. Time to sort right subarray}}\Big]$$

$\rho_p$ : احتمال اینکه p بعنوان اندیس عنصر محور برگردانده شود

$\alpha_p$ : میانگین تعداد تکرار عمل اصلی وقتیکه p اندیس عنصر محور باشد

$$A(n) = (n-1) + \frac{1}{n}\left(\sum_{p=1}^{n} A(p-1) + \sum_{q=1}^{n} A(q-1)\right) \qquad \boxed{q = n - p + 1}$$

$$A(n) = (n-1) + \frac{2}{n}\sum_{p=1}^{n} A(p-1)$$

# تحلیل پیچیدگی زمانی مرتب سازی سریع در حالت متوسط

$$A(n) = (n-1) + \frac{2}{n}\sum_{p=1}^{n} A(p-1)$$

دو طرف در n ضرب شود:

$$nA(n) = n(n-1) + 2\sum_{p=1}^{n} A(p-1)$$

بجای n قرار دهیم n-1:

$$(n-1)A(n-1) = (n-1)(n-2) + 2\sum_{p=1}^{n-1} A(p-1)$$

---

تفریق دو رابطه:

$$nA(n) - (n-1)A(n-1) = 2(n-1) + 2A(n-1)$$

ساده سازی:

$$\frac{1}{n+1}A(n) = \frac{1}{n}A(n-1) + 2\frac{n-1}{n(n+1)}$$

$$a_n = \frac{A(n)}{n+1}$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)}$$

# تحلیل پیچیدگی زمانی مرتب سازی سریع در حالت متوسط

- رابطه بازگشتی:

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)}$$
$$a_0 = 0$$

حل مشابه مثال B.22 کتاب

(با فرض n بزرگ : $\frac{2(n-1)}{n(n+1)} \approx \frac{2}{n}$ ):

$$a_n \approx 2\ln n$$

$$\downarrow \qquad \boxed{a_n = \frac{A(n)}{n+1}}$$

$$A_n \approx 2(n+1)\ln n \longrightarrow \boxed{A(n) \in \theta(n \lg n)}$$

- آیا در اینجا تحلیل حالت متوسط ارزشمند است؟

یادآوری: در کاربردهایی که الگوریتم به تعداد دفعات زیاد با ورودیهای گوناگون اجرا میشود (مانند یک الگوریتم مرتب سازی عمومی)، تحلیل حالت متوسط ارزشمند است.

# تحلیل پیچیدگی حافظه مرتب سازی سریع

- میزان حافظهٔ مصرفی در روال افراز؟

در روال افراز حافظه اضافی مصرف نمیشود.

```
void partition (index low, index high, index& pivotpoint)
{
    index i, j;
    keytype pivotitem;
    pivotitem = S[low];
    j = low;
    for(i=low+1; i<=high; i++)
        if (S[i] < pivotitem){
                j++;
                exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```

# تحلیل پیچیدگی حافظه مرتب سازی سریع

- آیا مرتب سازی سریع یک مرتب سازی درجا (**in-place**) است؟

گرچه حافظه اضافی ندارد، اما یک روش درجا نیز محسوب نمیشود، چون:

در فراخوانی بازگشتی برای مرتب سازی زیرآرایه اول، اندیسهای شروع و پایان زیرآرایه دوم (**pivotpoint** و **high**) باید در پشته نگهداری شوند.

- در بدترین حالت، یعنی هنگامی که همواره زیرآرایه دوم تهی باشد، تعداد **n-1** جفت اندیس در پشته قرار میگیرد.

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low){
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

- پیچیدگی حافظه مرتب سازی سریع در بدترین حالت:

$\theta(n)$ ← قابل تقلیل به $\theta(\lg n)$

# بهبود در مرتب سازی سریع

- تقلیل پیچیدگی حافظه:

اگر در تابع **quicksort** همواره اندیسهای مربوط به زیرآرایهٔ بزرگتر را در پشته قرار دهیم، آنگاه بدترین حالت مصرف حافظه زمانی است که هربار آرایه نصف شود. در اینحالت عمق پشته (حافظه مصرفی) **θ(lgn)** خواهد شد (سایر روشها در بخش 7.5 کتاب).

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low){
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

# بهبود در مرتب سازی سریع

- یک روش بهتر برای انتخاب عنصر محور:

  بجای عنصر اول، میانه سه عنصر اول، وسط، و آخر در هر مرحله بعنوان عنصر محور انتخاب شود (تضمین میشود که زیرآرایه ها تهی نباشند):

```
void partition (index low, index high, index& pivotpoint)
{
    index i, j;
    keytype pivotitem;
    pivotitem = S[low];
    j = low;
    for(i=low+1; i<=high; i++)
        if (S[i] < pivotitem){
                j++;
                exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```

pivotitem=median(S[low],S[(low+high)/2],S[high]);

# مقایسه مرتب سازی سریع و ادغامی

- مقایسه مرتب سازی سریع و ادغامی از لحاظ پیچیدگی زمانی:

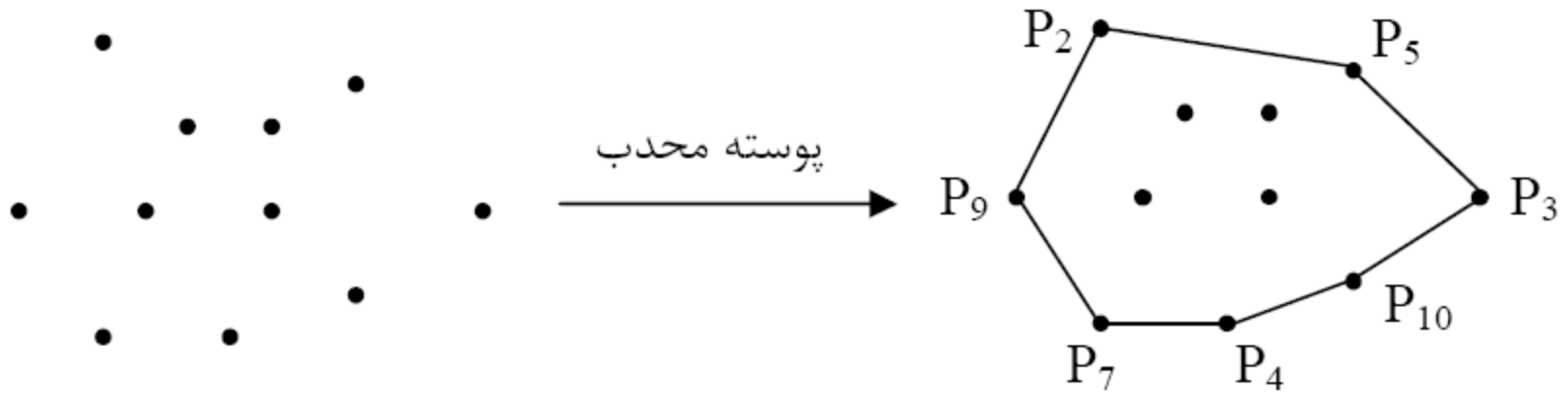
| پیچیدگی زمانی | بدترین حالت | حالت متوسط | بهترین حالت |
|---|---|---|---|
| مرتب سازی سریع | $\theta(n^2)$ | $\theta(n\lg n)$ | $\theta(n\lg n)$ |
| مرتب سازی ادغامی | $\theta(n\lg n)$ | $\theta(n\lg n)$ | $\theta(n\lg n)$ |

- مقایسه مرتب سازی سریع و ادغامی از لحاظ پیچیدگی حافظه:

| پیچیدگی حافظه | بدترین حالت |
|---|---|
| مرتب سازی سریع بهبود یافته | $\theta(\lg n)$ |
| مرتب سازی ادغامی بهبود یافته | $\theta(n)$ |

# مثال: یافتن پوسته محدب (Convex Hull)

- پوسته محدب (convex hull) برای یک مجموعه نقاط دو بعدی S: عبارت از کوچکترین چند ضلعی محدب است که شامل تمام نقاط S باشد.
- یک چندضلعی را محدب گوییم هرگاه برای هر دو نقطه دلخواه درون آن، پاره خط واصل آن دو نقطه بطور کامل درون چند ضلعی قرار گیرد.
- مثالی از پوسته محدب برای 11 نقطه:




$S=\{P_1, \ldots, P_{11}\}$

$CH(S)=\langle P_2, P_5, P_3, P_{10}, P_4, P_7, P_9\rangle$

# مثال: یافتن پوسته محدب (Convex Hull)

- میخواهیم الگوریتمی طراحی کنیم که با دریافت مختصات دو بعدی n نقطه یک مجموعه S بصورت $P_i=(x_i,y_i)$ ها $(i=1,\ldots,n)$، پوسته محدب آنها CH(S) را بصورت دنباله‌ای از رئوس (به ترتیب در جهت چرخش عقربه‌های ساعت) بدست آورده و در خروجی بدهد.
- یک الگوریتم بدیهی: جهت بررسی اینکه یک نقطه P از نقاط ورودی جزو رئوس پوسته محدب است یا نه، بصورت زیر عمل می‌کنیم:
  - تک تک سه تایی های ممکن از نقاط ورودی را در نظر گرفته و بررسی میکنیم که آیا نقطه P داخل مثلث متشکل از این سه نقطه هست یا نه؟
  - اگر P داخل چنین مثلثی باشد جزو رئوس پوسته محدب نیست، و در غیر اینصورت هست.
  - میدانیم بررسی وجود یک نقطه داخل یک مثلث در زمان ثابت $\Theta(1)$ قابل انجام است. لذا مرتبه این الگوریتم بدیهی میشود:

$$n\binom{n}{3} \in \theta(n^4)$$

- میخواهیم الگوریتمی از نوع تقسیم و حل طراحی کنیم که که با پیچیدگی زمانی کمتر از $n^4$ این کار را انجام دهد.

# الگوریتم Quick Hull

- ابتدا دو نقطه $P_1$ و $P_2$ از S را که به ترتیب کمترین و بیشترین مقدار x را دارند، تعیین میکنیم (فعلاً فرض میکنیم این نقاط منحصر به فرد هستند).
- چون $P_1$ و $P_2$ دو نقطه انتهایی هستند، حتماً جزو پوسته محدب هستند.
- مجموعه S به دو زیر مجموعه (دارای همپوشانی) $S_1$ و $S_2$ تقسیم میشود.
  مجموعه $S_1$ شامل تمام نقاط واقع در یک سمت از خط واصل $P_1$ و $P_2$ بعلاوه خود $P_1$ و $P_2$ بوده، و مجموعه $S_2$ شامل تمام نقاط واقع در سمت دیگر خط واصل $P_1$ و $P_2$ بعلاوه خود $P_1$ و $P_2$ می‌باشد.
- پوسته محدب مجموعه‌های $S_1$ و $S_2$ توسط روال بازگشتی Hull بدست می آیند.
- با داشتن پوسته محدب مجموعه‌های $S_1$ و $S_2$ ، پوسته محدب مجموعه S از اجتماع آنها بدست می‌آید.
- اگر دو نقطه $P_1$ و $P_2$ مذکور منحصر به فرد نباشند:
  - از میان نقاط با کمترین مقدار x آنهایی که کمترین و بیشترین مقدار y را دارند به ترتیب $P_1'$ و $P_1''$ مینامیم.

# الگوریتم Quick Hull

- بطور مشابه از میان نقاط با بیشترین مقدار x آنهایی که کمترین و بیشترین مقدار y را دارند به ترتیب $P_2'$ و $P_2''$ مینامیم.
- حال مجموعه $S_1$ شامل تمام نقاط واقع در سمت چپ خط واصل $P_1''$ و $P_2''$ بعلاوه خود $P_1''$ و $P_2''$ بوده، و مجموعه $S_2$ شامل تمام نقاط واقع در سمت راست خط واصل $P_1'$ و $P_2'$ بعلاوه خود $P_1'$ و $P_2'$ می‌باشد.

- روال بازگشتی Hull: یک الگوریتم تقسیم و حل است که برای یافتن پوسته محدب $S_1$ بصورت زیر عمل میکند:
  - از میان نقاط $S_1$ آن نقطه‌ای که با $P_1$ و $P_2$ مثلثی با بیشترین مساحت تشکیل میدهد را $P_3$ مینامد. این نقطه متعلق به پوسته محدب $S_1$ است. اگر منحصر بفرد نبود، نقطه‌ای که زاویه $P_3P_1P_2$ را بیشینه میکند انتخاب میشود.
  - شکستن مساله به دو بخش حاوی نقاطی که با حرکت از $P_1$ به $P_3$ در سمت چپ قرار میگیرند (بعلاوه خود $P_1$ و $P_3$)، و نقاطی که در سمت چپ $P_3$ به $P_2$ قرار میگیرند (بعلاوه خود $P_3$ و $P_2$). سایر نقاط داخلی بوده و لحاظ نمیشوند.
  - پوسته محدب هر یک از دو بخش فوق بصورت بازگشتی محاسبه میشود.
  - با قرار دادن نقاط یک پوسته به دنبال دیگری، پاسخها با هم ادغام میشوند.

# الگوریتم Quick Hull

- تحلیل پیچیدگی زمانی روال Hull: اگر $S_1$ شامل m نقطه باشد، یافتن $P_3$ ، شکستن $S_1$ به دو بخش، و ادغام دو پوسته محدب حاصل، هر یک در زمان $\Theta(m)$ انجام پذیر است:
  - رابطه بازگشتی برای زمان اجرا: $t(m)=t(m_1)+t(m_2)+\Theta(m)$ (که $m_1$ و $m_2$ تعداد نقاط دو بخش است و $m1+m2\leq m$).
  - مشابه با الگوریتم مرتب سازی سریع است.
  - پیچیدگی زمانی در بدترین حالت: $\Theta(m^2)$
  - پیچیدگی زمانی در حالت متوسط: $\Theta(mlgm)$
- تمرین: تحلیل و پیاده سازی الگوریتم Quick Hull

# در چه مواردی نباید از روش تقسیم و حل استفاده کرد

- اندازه زیرنمونه ها تقریباً برابر با اندازه نمونه اصلی باشد ← $\theta(c^n)$

مثال: الگوریتم بازگشتی محاسبه جمله n ام فیبوناچی:

**Fib(n)=Fib(n-1)+Fib(n-2)**

- تعداد زیرنمونه ها تقریباً برابر با اندازه نمونه اصلی باشد ← $\theta(n^{\lg n})$

مثال:

$$t(n) = nt(n/C) + g(n) \longrightarrow t(n) = \theta(n^{\log_C^n})$$

(البته برخی مسایل ذاتاً از مرتبه نمایی هستند. برای آنها میتوان از تقسیم و حل استفاده کرد، هرچند مرتبه نمایی خواهد بود. مثال: مسأله برجهای هانوی)