分治法

讲授者 王爱娟
aijuan321@foxmail.com

重庆理工大学 计算机科学与工程学院

August 22, 2024

目录

递归的概念

引例

1. 对于任意非负整数n，计算阶乘函数$F(n) = n!$的值；
2. 当$n > 1$时，$n! = 1 * 2 * ... * (n-1) * n = (n-1)! * n$；
3. 可以使用递归的方法计算$F(n) = F(n-1) * n$。

递归的概念

引例

1. 对于任意非负整数n，计算阶乘函数$F(n) = n!$的值；
2. 当$n > 1$时，$n! = 1 * 2 * \ldots * (n-1) * n = (n-1)! * n$；
3. 可以使用递归的方法计算$F(n) = F(n-1) * n$。

伪代码

```
function F(n)
    if n = 0 then
        return 1
    else
        return F(n − 1) ∗ n
    end if
end function
```

乘法执行次数$M(n)$

- $n > 0$时，$M(n) = M(n-1) + 1$
- 计算$F(n-1)$需要$M(n-1)$次乘法
- 计算$F(n-1) * n$需要1次乘法
- $n = 0$时，$M(0) = 0$，不需要乘法
- 替换法：$M(n) = M(n-i) + i = n$

递归方程求解 - 替换法

例1：计算$W(n)$，$W(n) = W(n-1) + n - 1, W(1) = 0$

$$\begin{aligned}
W(n) &= W(n-1) + n - 1 = W(n-2) + (n-2) + (n-1) \\
&= W(n-3) + (n-3) + (n-2) + (n-1) = \ldots\ldots \\
&= W(1) + 1 + 2 + \ldots + (n-2) + (n-1) = 1 + 2 + \ldots + (n-2) + (n-1) \\
&= n(n-1)/2
\end{aligned}$$

递归方程求解

递归方程求解 - 替换法

例1：计算$W(n)$，$W(n)=W(n-1)+n-1, W(1)=0$

$$
\begin{aligned}
W(n) &= W(n-1)+n-1 = W(n-2)+(n-2)+(n-1) \\
&= W(n-3)+(n-3)+(n-2)+(n-1) = \ldots\ldots \\
&= W(1)+1+2+\ldots+(n-2)+(n-1) = 1+2+\ldots+(n-2)+(n-1) \\
&= n(n-1)/2
\end{aligned}
$$

例2：计算$W(n)$，$W(n)=2W(n/2)+n-1, n=2^k, W(1)=0$

$$
\begin{aligned}
W(n) &= 2W(2^{k-1})+2^k-1 = 2[2W(2^{k-2})+2^{k-1}-1]+2^k-1 \\
&= 2^2W(2^{k-2})+2^k-2+2^k-1 \\
&= 2^2[2W(2^{k-3})+2^{k-2}-1]+2^k-2+2^k-1 \\
&= 2^3W(2^{k-3})+2^k-2^2+2^k-2+2^k-1 = \ldots\ldots \\
&= 2^kW(1)+k2^k-(2^{k-1}+2^{k-2}+\ldots+2+1) = k2^k-2^k+1 \\
&= nlogn-n+1
\end{aligned}
$$

递归分析方法

常见递归类型

减一算法

- 算法利用一个规模为n的实例和规模为$n-1$的给定实例之间的关系来对问题求解（插入排序，课本4.1）。
- $T(n) = T(n-1) + f(n)$

减常因子算法

- 规模为n的实例化简为一个规模为n/b的给定实例来求解（俄式乘法）。
- $T(n) = T(n/b) + f(n)$

分治算法

- 给定实例划分为若干个较小的实例，对每个实例递归求解，然后再把较小的实例合并成给定实例的一个解（快速排序，合并排序）。
- $T(n) = aT(n/b) + f(n)$

分治法的基本思想

基本常识

小问题往往比大问题容易解决！

分治法的基本思想

基本常识

小问题往往比大问题容易解决！

分治法的基本思想

1. 将规模为N的问题分解为k个规模较小的子问题，使这些子问题相互独立，可分别求解
2. 将k个子问题的解，合并成原问题的解
3. 如子问题的规模仍很大,则反复分解直到子问题小到可直接求解为止
4. 子问题的解法通常与原问题相同，导致递归过程

分治法的基本思想

例子

- 计算n个数字$a_1, a_2, ..., a_{n-1}, a_n$的和；
- 如果$n > 1$，可以把该问题划分为两个子问题，前一半的数字之和和后一半的数字之和；
- $a_1 + a_2 + ... + a_{n-1} + a_n = (a_1 + ... + a_{n/2-1}) + (a_{n/2} + a_{n/2+1} + ... + a_n)$
- 类似地一直划分，直到含有一个元素，直接返回该元素值；

分治法的时间效率分析

主定理(具体证明详见P376—-3 分治法)

$T(n)=a*T(\frac{n}{b})+f(n)$其中$T(1)=c, n=b^k, a\geq 1, b\geq 2, c>0$,
如果$f(n)\in\Theta(n^d), d\geq 0$,

1. if a < b^d, T(n) $\in \Theta(n^d)$
2. if a = b^d, T(n) $\in \Theta(n^d\log n)$
3. if a > b^d, T(n) $\in \Theta(n^{\log_b^a})$

注释:

1. $T(n)=aT(n/b)=a^2T(n/b^2)=a^{log_b n}T(1)=cn^{log_b a}$(课本附录A定理)
2. 本质上是比较$aT(n/b)$和$f(n)$增长速度，并利用P43定理；

分治法的时间效率分析

主定理应用例子

求解如下递归方程

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(n/2) + n$
3. $T(n) = T(n/3) + 1$

分治法的时间效率分析

主定理应用例子

求解如下递归方程

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(n/2) + n$
3. $T(n) = T(n/3) + 1$

答案

1. $a = 9, b = 3, f(n) = n \in \Theta(n)$，即$d = 1$，
   由于$a = 9 > b^d = 3^1$，所以$T(n) \in \Theta(n^{log_b^a}) = \Theta(n^2)$
2. $a = 1, b = 2, f(n) = n \in \Theta(n)$，即$d = 1$，
   由于$a = 1 < b^d = 2^1$，所以$T(n) = \Theta(n^d) = \Theta(n)$
3. $a = 1, b = 3, f(n) = 1 \in \Theta(1)$，即$d = 0$，
   由于$a = 1 = b^d = 3^0$，所以$T(n) \in \Theta(n^d logn) = \Theta(logn)$

合并排序

问题

将n个元素排成非递减顺序

合并排序

问题

将n个元素排成非递减顺序

解题思路

1. 若n为1,算法终止
2. 将n个待排元素分割成K个大致相等的子序列。这里令$K=2$，两个子序列分别为B和C
3. 分别对它们排序
4. 然后将B与C合并为一个有序序列

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B

1	2	6	9

C

4	5	7	8	13	16

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B

1	2	4	6	9

C

5	7	8	13	16

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B

1	2	4	5	6	9

C

7	8	13	16

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B

1	2	4	5	6	7	9

C

8	13	16

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B

1	2	4	5	6	7	8	9

C

13	16

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B

1	2	4	5	6	7	8	9	13

C

16

合并有序集合

例子：合并两个有序集B和C

B={1,2,6,9}, C={4,5,7,8,13,16}

合并过程

将C中的元素依次插入到B中的相应位置即可

B C

1	2	4	5	6	7	8	9	13	16

合并有序集合

merge(B,C,A)有序集合合并伪代码

```
i = 0; j = 0; k = 0
while i < len(B) and j < len(C) do
    if B[i] ≤ C[j] then
        A[k]= B[i]; i++
    else
        A[k]= C[j]; j++
    end if
    k = k + 1
end while
while i < len(B) do
    A[k] = B[i];k++; i++;
end while
while j < len(C) do
    A[k] = C[j]; k++; j++;
end while
```

▸ Skip mergesort alogirthm

merge的时间效率分析

如果两个集合B和C都是有序的，那么

1. merge算法返回的A也是有序的
2. 算法的基本操作是“比较”
3. 算法只需要遍历B一次，依次插入C的元素即可
4. 算法时间开销是$\Theta(n)$，效率高

merge的时间效率分析

如果两个集合B和C都是有序的，那么

1. merge算法返回的A也是有序的
2. 算法的基本操作是“比较”
3. 算法只需要遍历B一次，依次插入C的元素即可
4. 算法时间开销是$\Theta(n)$，效率高

合并排序的关键问题

- 规模为n的数组A划分为大小相等的两个集合B和C
- 如果B和C有序，则直接使用merge即可完成对A的排序
- 如何实现B和C有序

merge的时间效率分析

如果两个集合B和C都是有序的，那么

1. merge算法返回的A也是有序的
2. 算法的基本操作是“比较”
3. 算法只需要遍历B一次，依次插入C的元素即可
4. 算法时间开销是$\Theta(n)$，效率高

合并排序的关键问题

- 规模为n的数组A划分为大小相等的两个集合B和C
- 如果B和C有序，则直接使用merge即可完成对A的排序
- 如何实现B和C有序

递归地对B和C进行排序策略

常识

当数组长度为1时，数组天然有序

递归地对B和C进行排序策略

常识

当数组长度为1时，数组天然有序

对B和C进行排序策略

1. 对B和C进行不断的划分，产生更小的子集，直到子集的大小为1
2. 大小为1的子集天然有序，对它们采用merge方法合并，产生更大的有序集合
3. 依次对新产生的有序集合采用merge方法合并，最终可完成对B和C的排序，从而保证它们有序

例子

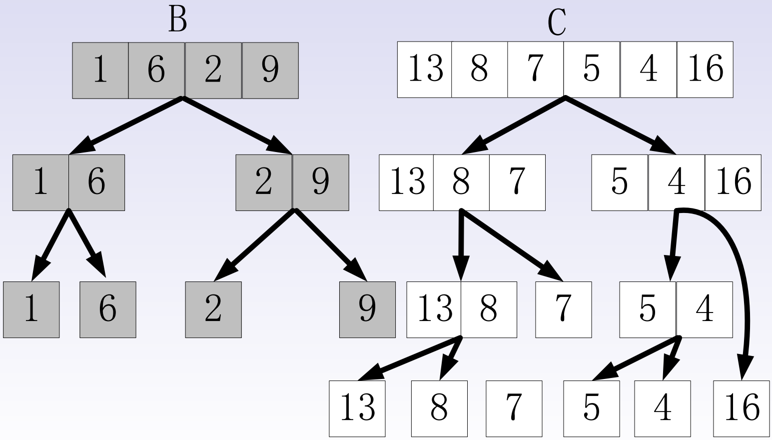
例子

$B = \{1,6,2,9\}$ $C = \{13,8,7,5,4,16\}$

例子

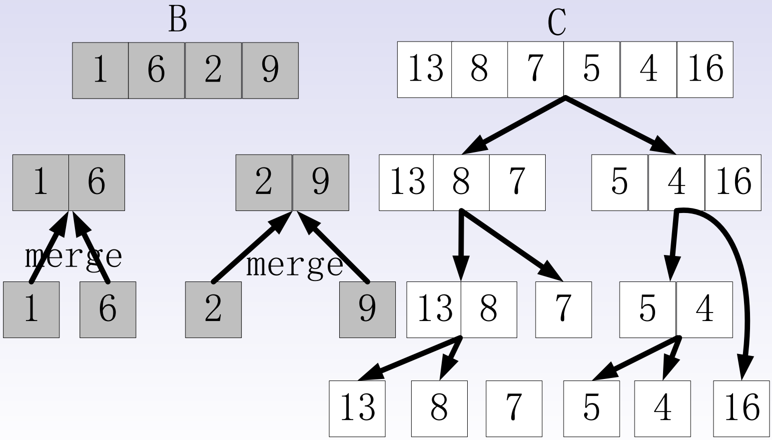
例子

B = {1,6,2,9} C = {13,8,7,5,4,16}



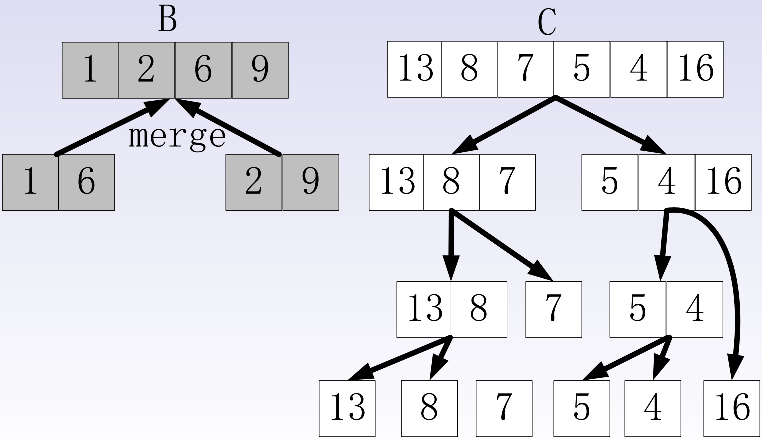
例子

B = {1,6,2,9} C = {13,8,7,5,4,16}



例子

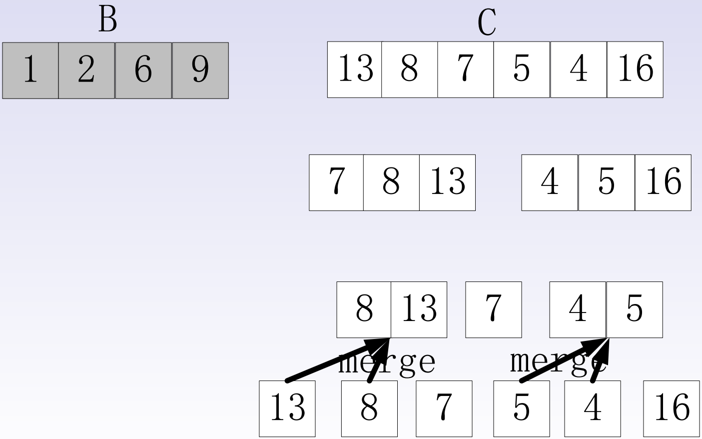
B = {1,6,2,9} C = {13,8,7,5,4,16}



例子

例子

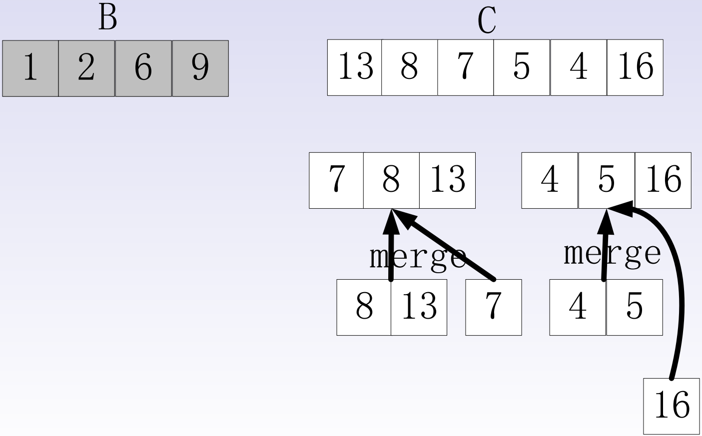
$B = \{1,6,2,9\}$ $C = \{13,8,7,5,4,16\}$



例子

例子

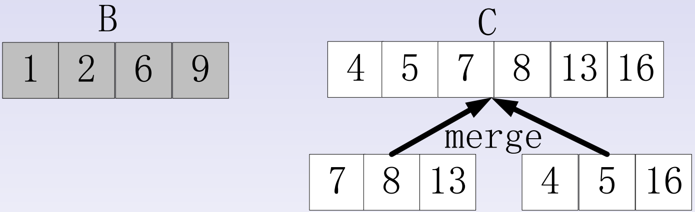
B = {1,6,2,9} C = {13,8,7,5,4,16}



例子

例子

B = {1,6,2,9} C = {13,8,7,5,4,16}



例子

例子

B = {1,6,2,9} C = {13,8,7,5,4,16}

B

1	2	6	9

C

4	5	7	8	13	16

归并排序过程总览

分治法

讲授者 王爱娟

目录

递归分析方法

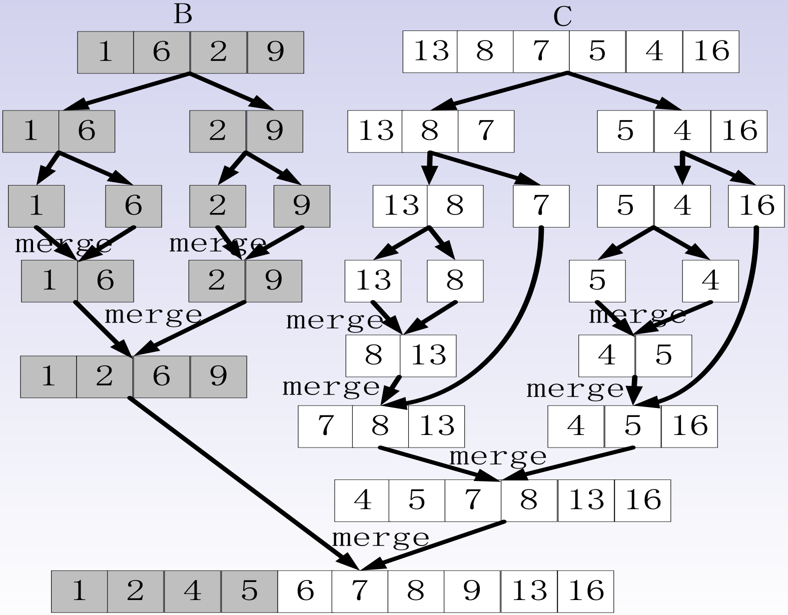
分治法的思想

合并排序

快速排序

堆和堆排序

大整数乘法和矩阵乘法



合并排序的递归算法

合并排序算法MergeSort(A[0..n-1])

//输入：未排序序列A[0..n-1]
//输出：已排序序列A[0..n-1]

if n > 1 **then**

copy A[0, $\lfloor \frac{n}{2} \rfloor$ - 1] to B[0, $\lfloor \frac{n}{2} \rfloor$ - 1]
copy A[$\lfloor \frac{n}{2} \rfloor$, n - 1] to C[0, $\lceil \frac{n}{2} \rceil$ - 1]
MergeSort(B[0, $\lfloor \frac{n}{2} \rfloor$ - 1])
MergeSort(C[0, $\lceil \frac{n}{2} \rceil$ - 1])
merge(B,C,A)

end if

Skip merge alogirthm

MergeSort算法解读

1. 数组A被对半划分为子数组B和C
2. B和C被递归地划分为更小的子集，直到子集大小为1
3. merge方法被调用合并有序的子集，直至所有集合被合并
4. 数组A排序完成

overview

MergeSort算法的时效分析

1. 算法的基本操作是"比较"
2. 设n $= 2^k$, 则关键字比较次数的递推关系式为

$$C(n) = 2C(\frac{n}{2}) + C_{merge}(n), \quad C(1) = 0$$

3. 在最坏情况下，merge的效率C_{merge}(n) = n-1,

$$C(n) = 2C(\frac{n}{2}) + n - 1, \quad C(1) = 0$$

$$\Rightarrow C(n) \in \Theta(n \log n)$$

theorem

当$n = 2^k$时，可以求得最差效率$C(n)$递推式的精确解：$C(n) = nlog_2 n - n + 1$

分治法之归并排序总结

问题

排序包含n个元素的数组A

- 问题规模的分解：将规模为n的数组A $\Rightarrow$ 两个规模为$\frac{n}{2}$的子问题B和C
- 继续分解：B和C的规模仍较大，继续划分它们为更小的子问题
- 停止分解：当子问题的规模为1时，停止划分
- 求子问题的解：由于规模为1的子问题，天然有序，无需对它们进行排序，它们本身就是所需的解
- 利用已有的解，求取更大问题的解：基于子集，采用一些操作(merge)，产生更大的有序集合
- 求取原问题的解：反复操作，直至所有子集被合并完成，即求得原问题的解！

▸ 分治思想

分治法之归并排序总结

问题

排序包含n个元素的数组A

- 问题规模的分解：将规模为n的数组A $\Rightarrow$ 两个规模为$\frac{n}{2}$的子问题B和C
- 继续分解：B和C的规模仍较大，继续划分它们为更小的子问题
- 停止分解：当子问题的规模为1时，停止划分
- 求子问题的解：由于规模为1的子问题，天然有序，无需对它们进行排序，它们本身就是所需的解
- 利用已有的解，求取更大问题的解：基于子集，采用一些操作(merge)，产生更大的有序集合
- 求取原问题的解：反复操作，直至所有子集被合并完成，即求得原问题的解！

分治思想

分治法之归并排序总结

问题

排序包含n个元素的数组A

- 问题规模的分解：将规模为n的数组A $\Rightarrow$ 两个规模为$\frac{n}{2}$的子问题B和C
- 继续分解：B和C的规模仍较大，继续划分它们为更小的子问题
- 停止分解：当子问题的规模为1时，停止划分
- 求子问题的解：由于规模为1的子问题，天然有序，无需对它们进行排序，它们本身就是所需的解
- 利用已有的解，求取更大问题的解：基于子集，采用一些操作(merge)，产生更大的有序集合
- 求取原问题的解：反复操作，直至所有子集被合并完成，即求得原问题的解！

分治思想

分治法之归并排序总结

问题

排序包含n个元素的数组A

- 问题规模的分解：将规模为n的数组A $\Rightarrow$ 两个规模为$\frac{n}{2}$的子问题B和C
- 继续分解：B和C的规模仍较大，继续划分它们为更小的子问题
- 停止分解：当子问题的规模为1时，停止划分
- 求子问题的解：由于规模为1的子问题，天然有序，无需对它们进行排序，它们本身就是所需的解
- 利用已有的解，求取更大问题的解：基于子集，采用一些操作(merge)，产生更大的有序集合
- 求取原问题的解：反复操作，直至所有子集被合并完成，即求得原问题的解！

分治思想

分治法之归并排序总结

问题

排序包含n个元素的数组A

- 问题规模的分解：将规模为n的数组A $\Rightarrow$ 两个规模为$\frac{n}{2}$的子问题B和C
- 继续分解：B和C的规模仍较大，继续划分它们为更小的子问题
- 停止分解：当子问题的规模为1时，停止划分
- 求子问题的解：由于规模为1的子问题，天然有序，无需对它们进行排序，它们本身就是所需的解
- 利用已有的解，求取更大问题的解：基于子集，采用一些操作(merge)，产生更大的有序集合
- 求取原问题的解：反复操作，直至所有子集被合并完成，即求得原问题的解！

分治思想

分治法之归并排序总结

问题

排序包含n个元素的数组A

- 问题规模的分解：将规模为n的数组A $\Rightarrow$ 两个规模为$\frac{n}{2}$的子问题B和C
- 继续分解：B和C的规模仍较大，继续划分它们为更小的子问题
- 停止分解：当子问题的规模为1时，停止划分
- 求子问题的解：由于规模为1的子问题，天然有序，无需对它们进行排序，它们本身就是所需的解
- 利用已有的解，求取更大问题的解：基于子集，采用一些操作(merge)，产生更大的有序集合
- 求取原问题的解：反复操作，直至所有子集被合并完成，即求得原问题的解！

分治思想

快速排序

回顾：中值问题

求一个数组(大小为n)的第k个最小元素。当k=$\lceil n/2 \rceil$时，该元素被称为中值。

解决办法：划分

- 将数组划分为两部分：一部分比某一值大，一部分比某一值小。
- 比如,{4,1,10,9,7,12,8,2,15} $\Rightarrow$ {1,2},{4},{9,7,12,8,10,15}
- 可见，通过划分发现4是数组A中的第三小元素，并且它正位于数组A中的第二位置,即A[2]=4。
- 这表明：一次划分可将一个数字放在(有序)数组A中的正确位置。

快速排序

回顾：中值问题

求一个数组(大小为n)的第k个最小元素。当k=$\lceil n/2 \rceil$时，该元素被称为中值。

解决办法：划分

- 将数组划分为两部分：一部分比某一值大，一部分比某一值小。
- 比如,{4,1,10,9,7,12,8,2,15} $\Rightarrow$ {1,2},{4},{9,7,12,8,10,15}
- 可见，通过划分发现4是数组A中的第三小元素，并且它正位于数组A中的第二位置,即A[2]=4。
- 这表明：一次划分可将一个数字放在(有序)数组A中的正确位置。

应用划分于排序

- 反复划分，就可确定所有元素在数组A中的正确位置

快速排序—基于划分的排序

算法思路

对于输入A[0.. n-1]，按以下二步骤进行排序：

1. 分区:取A中的一个元素P为支点(pivot)，将$A[0\cdots n\text{-}1]$划分成3段：$A[0\cdots s\text{-}1]$,P和$A[s+1\cdots n]$。
   - $P(=A[s])$，它在有序序列A的s位置(一个正确位置)
   - $A[0\cdots s\text{-}1]$, 它中任一元素$<P$
   - $A[s+1\cdots n\text{-}1]$, 它中的任一元素$>P$
2. 递归求解:分别递归地对$A[0\cdots s\text{-}1]$和$A[s+1\cdots n\text{-}1]$进行划分。
3. 终止条件:被划分的数组长度为1，即$A[0\cdots s\text{-}1]$的长度为1，$A[s+1\cdots n\text{-}1]$长度为1.

快速排序—基于划分的排序

算法思路

对于输入A[0.. n-1]，按以下二步骤进行排序：

1. 分区:取A中的一个元素P为支点(pivot)，将$A[0\cdots n\text{-}1]$划分成3段: $A[0\cdots s\text{-}1]$,P和$A[s+1\cdots n]$。
   - $P(=A[s])$，它在有序序列A的s位置(一个正确位置)
   - $A[0\cdots s\text{-}1]$, 它中任一元素<P
   - $A[s+1\cdots n\text{-}1]$, 它中的任一元素>P
2. 递归求解:分别递归地对$A[0\cdots s\text{-}1]$和$A[s+1\cdots n\text{-}1]$进行划分。
3. 终止条件:被划分的数组长度为1，即$A[0\cdots s\text{-}1]$的长度为1，$A[s+1\cdots n\text{-}1]$长度为1.

快速排序—基于划分的排序

算法思路

对于输入A[0.. n-1]，按以下二步骤进行排序：

1. 分区:取A中的一个元素P为支点(pivot)，将A[0···n-1]划分成3段：A[0···s-1],P和A[s+1···n]。
   - P(=A[s])，它在有序序列A的s位置(一个正确位置)
   - A[0···s-1], 它中任一元素<P
   - A[s+1···n-1], 它中的任一元素>P
2. 递归求解:分别递归地对A[0···s-1]和A[s+1···n-1]进行划分。
3. 终止条件:被划分的数组长度为1，即A[0···s-1]的长度为1，A[s+1···n-1]长度为1.

快速排序—基于划分的排序

算法思路

对于输入A[0.. n-1]，按以下二步骤进行排序：

1. 分区:取A中的一个元素P为支点(pivot)，将$A[0\cdots n\text{-}1]$划分成3段：$A[0\cdots s\text{-}1]$,P和$A[s+1\cdots n]$。
   - P(=A[s])，它在有序序列A的s位置(一个正确位置)
   - $A[0\cdots s\text{-}1]$, 它中任一元素<P
   - $A[s+1\cdots n\text{-}1]$, 它中的任一元素>P
2. 递归求解:分别递归地对$A[0\cdots s\text{-}1]$和$A[s+1\cdots n\text{-}1]$进行划分。
3. 终止条件:被划分的数组长度为1，即$A[0\cdots s\text{-}1]$的长度为1，$A[s+1\cdots n\text{-}1]$长度为1.

快速排序—基于划分的排序

算法思路

对于输入A[0.. n-1]，按以下二步骤进行排序：

1. 分区:取A中的一个元素P为支点(pivot)，将A[0⋯n-1]划分成3段：A[0⋯s-1],P和A[s+1⋯n]。
   - P(=A[s])，它在有序序列A的s位置(一个正确位置)
   - A[0⋯s-1], 它中任一元素<P
   - A[s+1⋯n-1], 它中的任一元素>P
2. 递归求解:分别递归地对A[0⋯s-1]和A[s+1⋯n-1]进行划分。
3. 终止条件:被划分的数组长度为1，即A[0⋯s-1]的长度为1，A[s+1⋯n-1]长度为1.

快速排序—基于划分的排序

算法思路

对于输入A[0.. n-1]，按以下二步骤进行排序：

1. 分区:取A中的一个元素P为支点(pivot)，将A[0⋯n-1]划分成3段：A[0⋯s-1],P和A[s+1⋯n]。
   - P(=A[s])，它在有序序列A的s位置(一个正确位置)
   - A[0⋯s-1], 它中任一元素<P
   - A[s+1⋯n-1], 它中的任一元素>P
2. 递归求解:分别递归地对A[0⋯s-1]和A[s+1⋯n-1]进行划分。
3. 终止条件:被划分的数组长度为1，即A[0⋯s-1]的长度为1，A[s+1⋯n-1]长度为1.

例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序

例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序

4	1	10	9	7	12	8	2	15

例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序

4	1	10	9	7	12	8	2	15
2	1	4	10	9	7	12	8	15

例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序

4 1 10 9 7 12 8 2 15

2 1 4 10 9 7 12 8 15

10 9 7 12 8 15

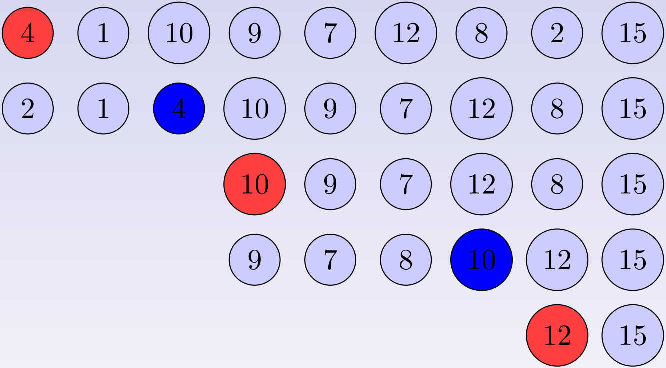
例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序

4	1	10	9	7	12	8	2	15
2	1	4	10	9	7	12	8	15
			10	9	7	12	8	15
			9	7	8	10	12	15

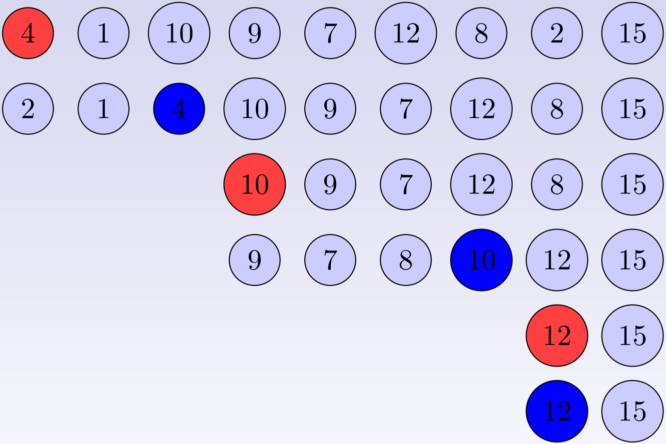
例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序



例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序



例子

采用划分的方式对数组A={4,1,10,9,7,12,8,2,15}进行排序

4 1 10 9 7 12 8 2 15

2 1 4 10 9 7 12 8 15

10 9 7 12 8 15

9 7 8 10 12 15

12 15

12 15

可以看出，每一次划分都可确定一个元素的位置

快速排序算法

快速排序算法QuickSort(A[l···r])

//使用快速排序法对序列或者子序列排序
//输入：子序列A[l..r]或者序列本身A[0..n-1]
//输出：非递减序列A

if l < r **then**
  s = partition(A[l···r])
  QuickSort(A[l···s-1])
  QuickSort(A[s+1···r])
end if

快速排序算法

快速排序算法QuickSort(A[l···r])

//使用快速排序法对序列或者子序列排序
//输入：子序列A[l..r]或者序列本身A[0..n-1]
//输出：非递减序列A

if l < r **then**
  s = partition(A[l···r])
  QuickSort(A[l···s-1])
  QuickSort(A[s+1···r])
end if

partition：快速排序算法的核心

上述代码表明，快速排序算法的核心是partition，即如何对数组A进行划分！

回顾：Lomuto划分

Lomuto划分：单向扫描

- 从左至右地扫描数组，在扫描过程中，将每个元素与P比较
- 若当前元素小于P，则置于数组前段，否则置于后端
- 扫描完后，数组A划分为两个连续区域：小于P的区域和大于等于P的区域
- s的位置特点：A[s] $<$ P, A[s+1] $\geq$ P
- 交换A[s]与A[l]，即完成划分
- A[s]正位于一个有序的数组A的正确位置

l r

P	?

回顾：Lomuto划分

Lomuto划分：单向扫描

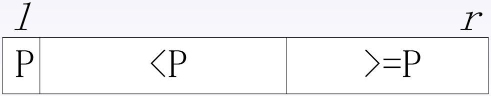
- 从左至右地扫描数组，在扫描过程中，将每个元素与P比较
- 若当前元素小于P，则置于数组前段，否则置于后端
- 扫描完后，数组A划分为两个连续区域：小于P的区域和大于等于P的区域
- s的位置特点：A[s] $<$ P, A[s+1] $\geq$ P
- 交换A[s]与A[l]，即完成划分
- A[s]正位于一个有序的数组A的正确位置

l			r
P	<P	>=P	?

回顾：Lomuto划分

Lomuto划分：单向扫描

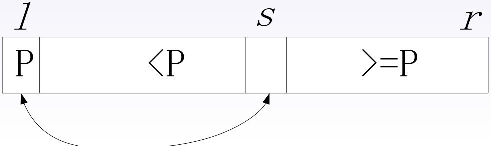
- 从左至右地扫描数组，在扫描过程中，将每个元素与P比较
- 若当前元素小于P，则置于数组前段，否则置于后端
- 扫描完后，数组A划分为两个连续区域：小于P的区域和大于等于P的区域
- s的位置特点：$A[s] < P$, $A[s+1] \geq P$
- 交换A[s]与A[l]，即完成划分
- A[s]正位于一个有序的数组A的正确位置



回顾：Lomuto划分

Lomuto划分：单向扫描

- 从左至右地扫描数组，在扫描过程中，将每个元素与P比较
- 若当前元素小于P，则置于数组前段，否则置于后端
- 扫描完后，数组A划分为两个连续区域：小于P的区域和大于等于P的区域
- s的位置特点：$A[s] < P$, $A[s+1] \geq P$
- 交换A[s]与A[l]，即完成划分
- A[s]正位于一个有序的数组A的正确位置



回顾：Lomuto划分

Lomuto划分：单向扫描

- 从左至右地扫描数组，在扫描过程中，将每个元素与P比较
- 若当前元素小于P，则置于数组前段，否则置于后端
- 扫描完后，数组A划分为两个连续区域：小于P的区域和大于等于P的区域
- s的位置特点：A[s] $<$ P, A[s+1] $\geq$ P
- 交换A[s]与A[l]，即完成划分
- A[s]正位于一个有序的数组A的正确位置

l		s	r
A[s]	<P	P	>=P

回顾：Lomuto划分

Lomuto划分：单向扫描

- 从左至右地扫描数组，在扫描过程中，将每个元素与P比较
- 若当前元素小于P，则置于数组前段，否则置于后端
- 扫描完后，数组A划分为两个连续区域：小于P的区域和大于等于P的区域
- s的位置特点：$A[s] < P$, $A[s+1] \geq P$
- 交换A[s]与A[l]，即完成划分
- A[s]正位于一个有序的数组A的正确位置

快速排序的划分方法

快速排序的划分：双向扫描

- 指针j从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素A[j]时停止
- 指针i从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素A[i]时停止
- 若指针不相交，交换A[i]和A[j]。
- 然后继续各自的扫描。

快速排序的划分方法

快速排序的划分：双向扫描

- 指针j从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素A[j]时停止
- 指针i从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素A[i]时停止
- 若指针不相交，交换A[i]和A[j]。
- 然后继续各自的扫描。

快速排序的划分方法

快速排序的划分：双向扫描

- 指针j从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素A[j]时停止
- 指针i从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素A[i]时停止
- 若指针不相交，交换A[i]和A[j]。
- 然后继续各自的扫描。

快速排序的划分方法

快速排序的划分：双向扫描

- 指针j从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素A[j]时停止
- 指针i从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素A[i]时停止
- 若指针不相交，交换A[i]和A[j]。
- 然后继续各自的扫描。

快速排序的划分方法

快速排序的划分：双向扫描

- 指针j从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素A[j]时停止
- 指针i从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素A[i]时停止
- 若指针不相交，交换A[i]和A[j]。
- 然后继续各自的扫描。

快速排序的划分方法

快速排序的划分：双向扫描

- 指针j从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素A[j]时停止
- 指针i从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素A[i]时停止
- 若指针不相交，交换A[i]和A[j]。
- 然后继续各自的扫描。

两种扫描方式的比较

- Lomuto划分是单向扫描：从左至右扫描数组
- 快速排序使用的划分是双向扫描，但先从右端开始扫描，然后在从左端开始扫描，效率更高

快速排序的划分方法的例子

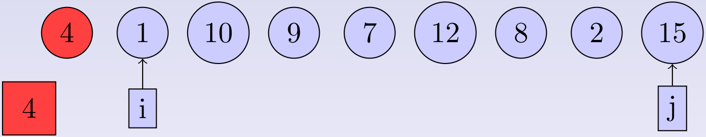
例子

采用双向扫描的方式对数组A=$\{4,1,10,9,7,12,8,2,15\}$进行划分

快速排序的划分方法的例子

例子

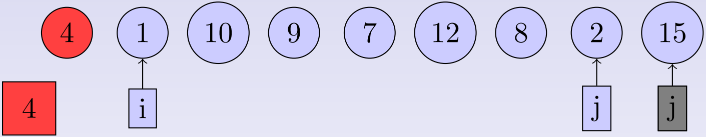
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

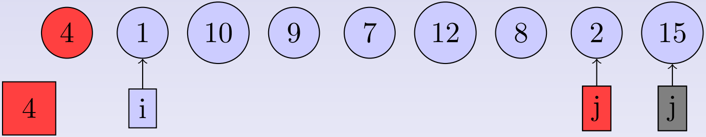
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

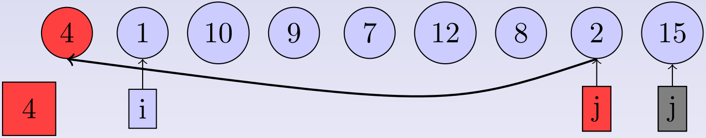
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

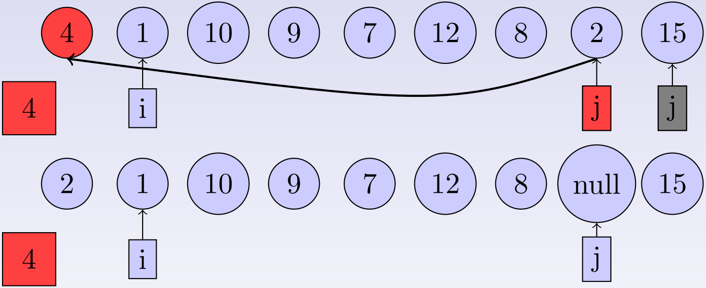
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

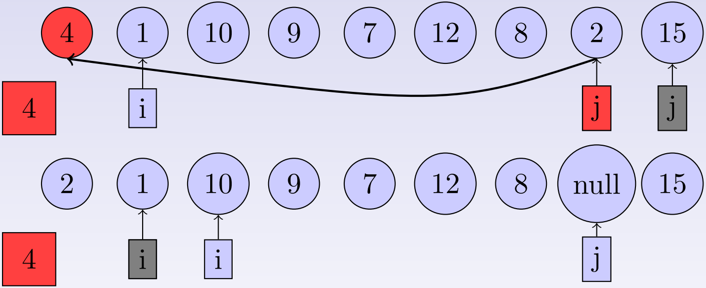
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

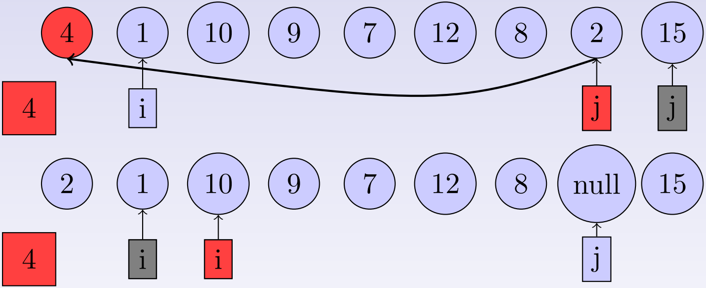
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

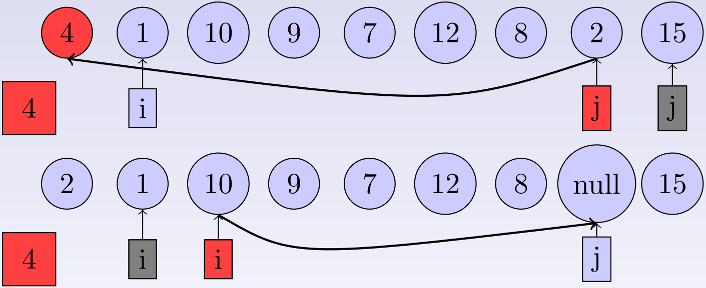
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

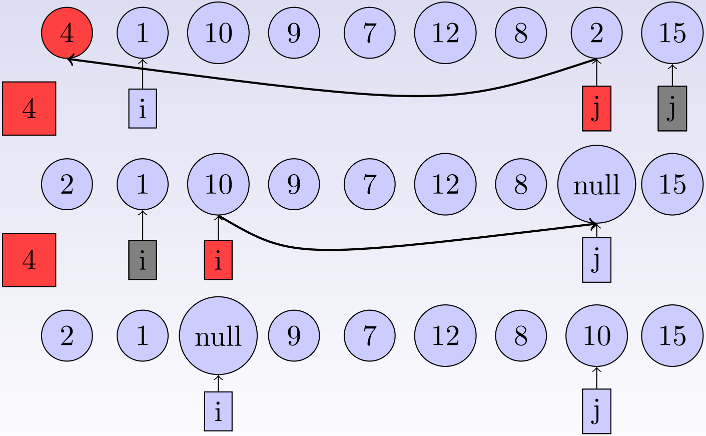
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

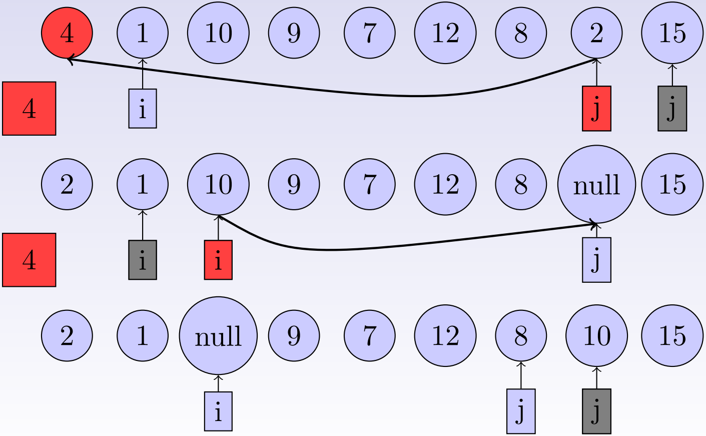
例子

采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



例子

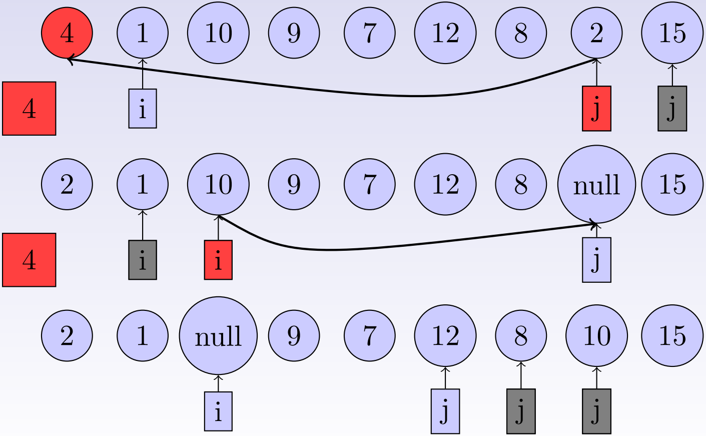
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

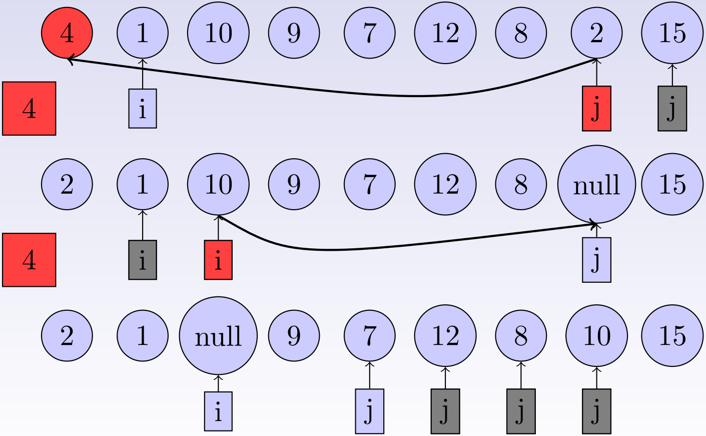
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

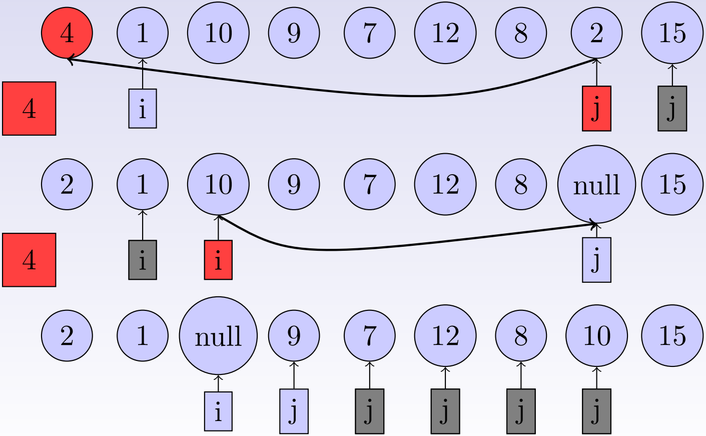
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

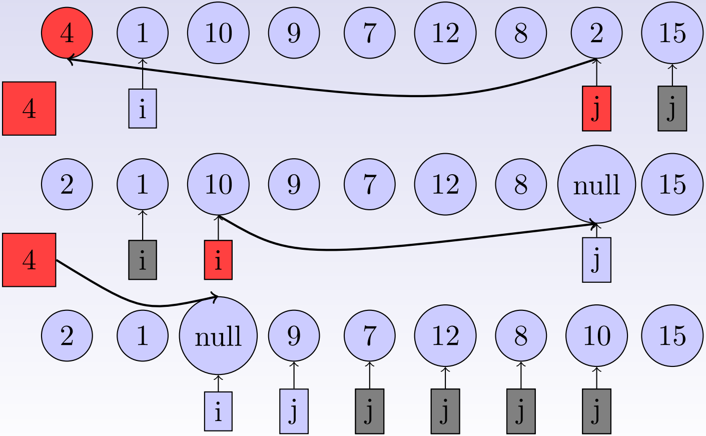
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

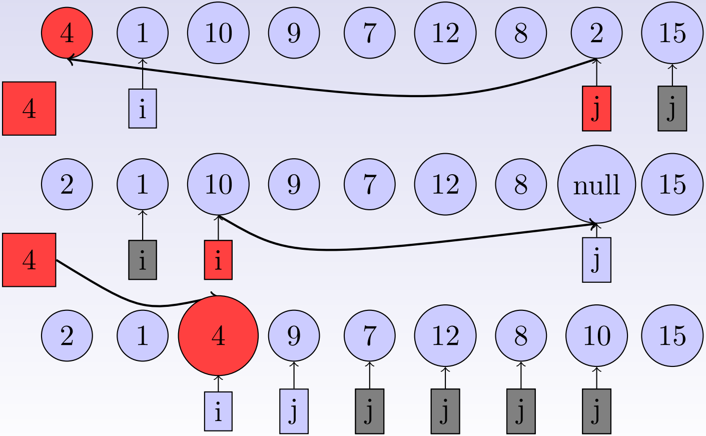
采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分方法的例子

例子

采用双向扫描的方式对数组A={4,1,10,9,7,12,8,2,15}进行划分



快速排序的划分算法

partition算法

//输入：数组A[low,$\cdots$,high]
//输出：划分数组A

```
i = low; j = high; x = A[low]
while i < j do
    while i < j and A[j] > x do
        j = j - 1;
    end while
    A[i] = A[j]; i=i+1
    while i < j and A[i] < x do
        i = i + 1;
    end while
    A[j] = A[i]; j=j-1
end while
A[i] = x;
```

快速排序效率分析

最坏情况下的效率分析

- 最差情况：所有分裂点都趋于极端：两个子数组中，有一个为空，另一个子数组仅仅比被划分数组少了一个元素
- 基本操作：比较
- 考虑一个严格递增的数组
- 进行了n+1次比较后，数组A[0,···,n-1]被划分为P（A[0]）和A[1,···,n-1]
- 进行了n次比较后，数组A[1,···,n-1]被划分为A[1]和A[2,···,n-1]
- 如此反复，直到最后一个子数组A[n-2,n-1]，进行3次比较后结束
- 总的比较次数：

$$C_{worst}(n) = (n+1)+n+\cdots+3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$$

快速排序效率分析

最好情况下的效率分析

- 最优情况：所有分裂点位于相应子数组的中点
- 基本操作：比较
- 进行了n次比较后，数组A[0,$\cdots$,n-1]被划分为P和两个大小近似相当的部分：A[0,$\cdots$,s-1]和A[s+1,$\cdots$,n-1]
- 如此反复，直至对数组的划分结束
- 总的比较次数：

$$C_{best}(n) = 2 * C_{best}(\frac{n}{2}) + n \in \Theta(n \log n)$$

快速排序效率分析

平均情况下的效率分析

- 基本操作：比较
- 总的比较次数：

$$C_{avg}(n) = \frac{1}{n}\Sigma_{s=0}^{n-1}[n+1+C_{avg}(s)+C_{avg}(n-1-s)],$$

$$C_{avg}(0) = 0, C_{avg}(1) = 0,$$

$$\Rightarrow C_{avg}(n) \approx 2n\ln n \approx 1.39n\log_2 n \in \Theta(n\log n)$$

堆和堆排序

堆

堆可以定义为一棵二叉树，并且满足下面的条件：
完备性（完全二叉树）：除了最后一层最右边的元素可能缺，其他都是满的；
堆特性：每个节点的值大于等于它的子女节点的值，且只存在一棵n个节点的完全二叉树，它的高度等于$\lfloor log_2 n \rfloor$；

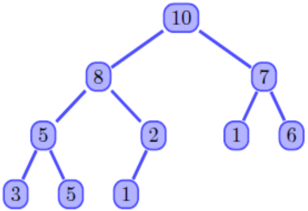
堆操作与堆构造

插入操作：与自顶向下构造堆相似，最坏情况下的复杂度为$O(log_2 n)$（树的高度）
删除操作：要删除的节点和堆的最后一个节点交换，然后删除；
插入、删除等操作复杂度与堆构造相同。

堆排序

堆的特性和实现

- 只存在一棵n个节点的完全二叉树，它的高度等于$log_2 n$；
- 可以用数据实现堆，从上到下，从左到右的方式记录堆的元素；
- 为了表达方便，在数组的开始位置设置为空；
- 父母节点i位于数组的前$\lfloor n/2 \rfloor$个位置，它的子女位于$2i$和$2i+1$；
- 对于一个位于$i(2 \leqslant i \leqslant n)$的建来说，它的父母将会位于$\lfloor i/2 \rfloor$。



堆的数组表示

0	1	2	3	4	5
	10	8	7	5	2

6	7	8	9	10
1	6	3	5	1

堆的构造

堆的构造–自底向上:

方法要点

1. 按照给定的顺序放置节点;
2. 从最后的父母节点K开始，到根为止，检查节点是否满足要求；如果不满足，交换K和子女最大键值节点的位置;
3. 检查新的位置上，执行步骤2，检查键值K是不是满足要求;
4. 重复步骤2和3，直到对树的根完成操作。

实例：对列表2, 9, 7, 6, 5, 8构造堆

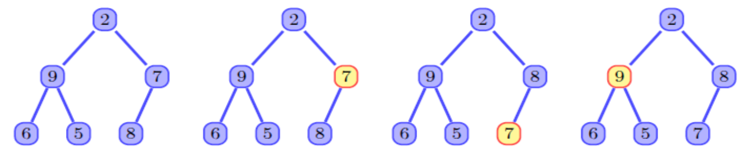
堆的构造

堆的构造–自底向上:

方法要点

1. 按照给定的顺序放置节点;
2. 从最后的父母节点K开始，到根为止，检查节点是否满足要求；如果不满足，交换K和子女最大键值节点的位置;
3. 检查新的位置上，执行步骤2，检查键值K是不是满足要求;
4. 重复步骤2和3，直到对树的根完成操作。

实例：对列表2, 9, 7, 6, 5, 8构造堆



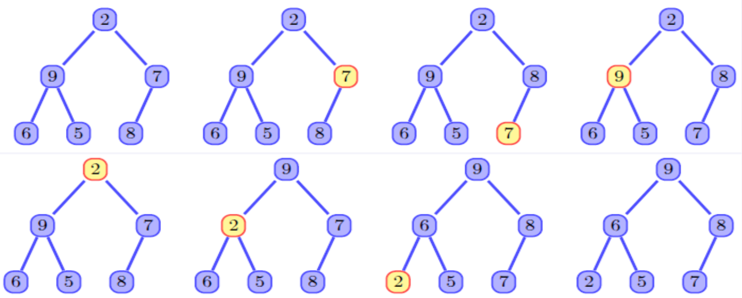
堆的构造

堆的构造–自底向上:

方法要点

1. 按照给定的顺序放置节点;
2. 从最后的父母节点K开始，到根为止，检查节点是否满足要求；如果不满足，交换K和子女最大键值节点的位置;
3. 检查新的位置上，执行步骤2，检查键值K是不是满足要求;
4. 重复步骤2和3，直到对树的根完成操作。

实例：对列表2, 9, 7, 6, 5, 8构造堆



堆排序

堆排序算法

第一步：构造堆；
第二步：删除最大键值，即对剩下的堆应用$n-1$次根删除操作。
重复一二步，最终结果就是按照降序删除了的数据元素。

时间效率分析

堆构造阶段的时间效率属于$O(n)$；
第二阶段就是把堆的规模从n消减到2的过程，消根所需要的键值比较次数为$C(n) \leq 2nlog_2n$；
堆排序整个过程的时间效率为$O(nlogn)$。

为什么需要大整数

- int型变量的存储空间为32位，因此变量的取值范围在-2^{31}~2^{31}(-2,147,483,648~2,147,483,647)之间
- long型变量的存储空间为64位，因此变量的取值范围在-2^{63}~2^{63}(-9,223,372,036,854,775,808~9,223,372,036,854,775,807)之间
- 两个100位的十进制数相乘，能否使用整数int型或，long型表示？
- 不行，$10^{100} >> 2^{63}$

大整数乘法

为什么需要大整数

- int型变量的存储空间为32位，因此变量的取值范围在-2^{31}～2^{31}(-2,147,483,648～2,147,483,647)之间
- long型变量的存储空间为64位，因此变量的取值范围在-2^{63}～2^{63}(-9,223,372,036,854,775,808～9,223,372,036,854,775,807)之间
- 两个100位的十进制数相乘，能否使用整数int型或，long型表示？
- 不行，$10^{100} >> 2^{63}$

大整数乘法

为什么需要大整数

- int型变量的存储空间为32位，因此变量的取值范围在-2^{31}~2^{31}(-2,147,483,648~2,147,483,647)之间
- long型变量的存储空间为64位，因此变量的取值范围在-2^{63}~2^{63}(-9,223,372,036,854,775,808~9,223,372,036,854,775,807)之间
- 两个100位的十进制数相乘，能否使用整数int型或，long型表示?
- 不行，$10^{100} >> 2^{63}$

大整数乘法

为什么需要大整数

- int型变量的存储空间为32位，因此变量的取值范围在$-2^{31} \sim 2^{31}$(-2,147,483,648~2,147,483,647)之间
- long型变量的存储空间为64位，因此变量的取值范围在$-2^{63} \sim 2^{63}$(-9,223,372,036,854,775,808~9,223,372,036,854,775,807)之间
- 两个100位的十进制数相乘，能否使用整数int型或，long型表示？
- 不行，$10^{100} >> 2^{63}$

大整数乘法问题

- 某些应用中，如当代的密码技术，需要计算超过上千位的二进制的乘法；
- 假设X和Y是两个n位的二进制数，$n = 2^k$，计算XY.

解决办法

- 按照通常的做法，需要总共n^2次乘法计算；
- 考虑分治法，将X和Y分成相等的两段，每段$n/2$位，即

$$X = A2^{n/2} + B$$

$$Y = C2^{n/2} + D$$

$$XY = AC2^n + (AD + BC)2^{n/2} + BD$$

- 规模为n的原问题转换为4个规模为$n/2$的子问题；

大整数乘法

分治法求解

复杂度分析:

- 计算需要的乘法次数递归方程:

$$W(n) = \begin{cases} 4W(n/2), \ n > 1 \\ 1, \ n = 1 \end{cases}$$

- 根据主定理可得到$W(n) \in \Theta(n^2)$.

注意:

- 虽然采用了分治法，但是时间复杂度并没有降低;
- 回顾主定理，当问题规模减半$b = 2$，合并（加减法运算次数）的复杂度为$\Theta(n)$时，子问题数$a > 2$ 时，时间复杂度为$\Theta(n^{log_b a})$，减少子问题数可降低时间复杂度。

大整数乘法

改进的分治法

改进思路：

- $AD + BC = (A - B)(D - C) + AC + BD$
- AC和BD已知，那么子问题数目从原来的4个变为3个；

大整数乘法

改进的分治法

改进思路：

- $AD + BC = (A - B)(D - C) + AC + BD$
- AC和BD已知，那么子问题数目从原来的4个变为3个；

时间复杂度分析

- 乘法次数递归方程：$M(n) = 3M(n/2), n > 1; M(1) = 1;$
- 根据主定理可得到$M(n) \in \Theta(n^{log_2 3}) \approx (n^{1.59})$;
- 合并子问题的复杂度：6次规模为n的整数加减法操作，可记为cn;
- 加减次数递归方程：$A(n) = 3A(n/2) + cn, n > 1; A(1) = 0;$
- 根据主定理可得到$A(n) \in \Theta(n^{log_2 3}) \approx (n^{1.59})$;
- 算法总的时间效率为$\Theta(n^{1.59})$，效率有明显的提升；

矩阵乘法

矩阵相乘问题

假设A和B是两个n阶的矩阵，$n=2^k$，计算$C=AB$;

解决办法

- $C_{ij}=a_{i1}b_{1j}+a_{i2}b_{2j}+...+a_{in}b_{nj}$
- 计算C_{ij}需要n次乘法(不考虑加法)，计算C需要n^3次乘法；
- 考虑分治法：

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

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

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

- 规模为n的原问题转换为8个规模为$n/2$的子问题；

矩阵乘法

分治法求解

复杂度分析

- 乘法次数递归方程：

$$M(n) = \begin{cases} 8M(n/2), \ n > 1 \\ 1, \ n = 1 \end{cases}$$

- 根据主定理可得到$M(n) \in \Theta(n^{log8}) = \Theta(n^3)$

注意：

- 与传统方法比，时间复杂度并没有降低；
- 同样地，考虑减少子问题数以降低时间复杂度。

Strassen (施特拉森)矩阵乘法

Strassen (施特拉森)矩阵乘法

- 分块矩阵相乘:

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

- 中间结果:

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

- 计算最终结果:

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

时间复杂度分析

- 合并子问题的复杂度：矩阵加法（$(n/2)^2$个元素相加）18次；
- 乘法计算总次数递归方程：

$$M(n) = \begin{cases} 7M(n/2), \ n > 1 \\ 1, \ n = 1 \end{cases}$$

- 根据主定理可得到$M(n) \in \Theta(n^{log_2 7}) \approx (n^{2.807})$
- 加法计算总次数递归方程：

$$A(n) = \begin{cases} 7A(n/2) + 18(n/2)^2, \ n > 1 \\ 0, \ n = 1 \end{cases}$$

- 根据主定理可得到$A(n) \in \Theta(n^{log_2 7}) \approx \Theta(n^{2.807})$
- 算法总的时间效率为$\Theta(n^{2.807})$，效率有明显的提升；

作业

(1) 习题5.1：**8, 9**
(2) 习题5.4：**7**
(3) 习题5.2：**11** (查找教材最后的英文文献[Raw91]，阅读相关内容)