算法基础HW3

PB19071535徐昊天

问题1

答：

(a)$T(n)=\Theta(nlog(logn))$

证明如下：

将递归式展开有：

$$
\begin{aligned}
T(n)&=2T(\frac{n}{2})+\frac{n}{logn}\\
&=2\times 2T(\frac{n}{4})+\frac{n}{logn}+2\times\frac{n/2}{log(n/2)}\\
&=2\times 2\times 2T(\frac{n}{8})+\frac{n}{logn}+2\times\frac{n/2}{log(n/2)}+2\times 2\times\frac{n/4}{log(n/4)}\\
&=\dots\\
&=2^iT(\frac{n}{2^i})+n\sum_{k=0}^{i-1}\frac{1}{log\frac{n}{2^k}}\\
&=2^iT(\frac{n}{2^i})+n\sum_{k=0}^{i-1}\frac{1}{logn-k}\quad(\text{展开结束时有}n=2^i)\\
&=nT(1)+n\sum_{k=0}^{i-1}\frac{1}{i-k}\\
&=nT(1)+n\sum_{k=1}^{i}\frac{1}{k}\\
&\text{根据微积分的知识：}\sum_{k=1}^{i}\frac{1}{k+1}\le\int_1^i\frac{1}{k}dx=ln(i)\le\sum_{k=1}^{i}\frac{1}{k}\\
&\text{可知}\sum_{k=1}^{i}\frac{1}{k}=\Theta(logi)\\
&\text{故}T(n)=\Theta(nlogi)=\Theta(nlog(logn))
\end{aligned}
$$

由上，原命题得证。

(b)$T(n)=\Theta(n(logn)^2)$

证明如下：

令$n=4^k$，代入递归式则有：$T(4^k)=4T(4^{k-1})+4^k\times log(4^k)$

即$T(4^k)=4T(4^{k-1})+4^k\times 2k$

令$S(k)=T(4^k)$，展开递归式有：

$$
\begin{aligned}
S(k) &= 4S(k-1) + 4^k \times 2k \\
&= 4(4S(k-2) + 2(k-1) \times 4^{k-1}) + 4^k \times 2k \\
&= 16S(k-2) + 2(k-1) \times 4^k + 2k \times 4^k \\
&= 16(4S(k-3) + 2(k-2) \times 4^{k-2}) + 2(k-1) \times 4^k + 2k \times 4 \\
&= 64S(k-3) + 2(k-2) \times 4^k + 2(k-1) \times 4^k + 2k \times 4^k \\
&= \dots \\
&= 4^k S(0) + 4^k \times 2(1+2+\dots+k) \\
&= 4^k S(0) + 4^k \times k(k+1)
\end{aligned}
$$

将S函数转化为T函数作分析：

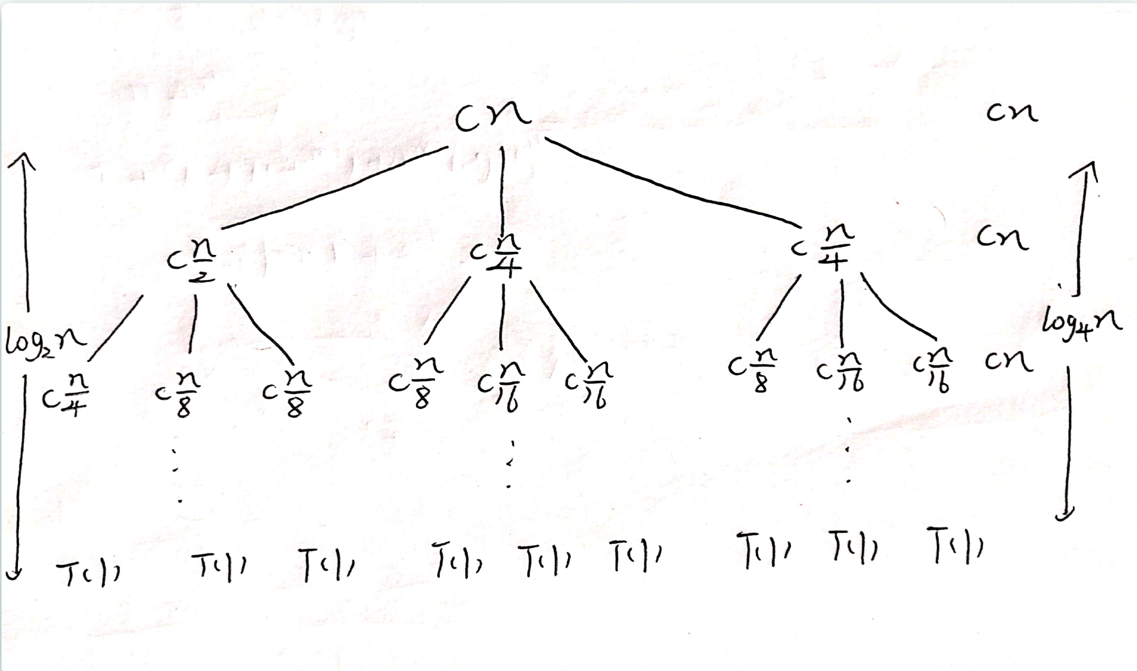
$$
\begin{gathered}
T(n) = nT(1) + n \times log_4 n(log_4 n + 1) \\
=> T(n) = \Theta(n(logn)^2)
\end{gathered}
$$

由上，原命题得证。

(c)$T(n) = \Theta(nlogn)$

证明如下：

根据递归式构造递归树如下：



显然，递归树每一层的代价均为cn，且递归树两侧高度不同，左侧高度较高，为$log_2 n + 1$，右侧高度较低，为$log_4 n + 1$。

故有$cn(log_4 n + 1) \le T(n) \le cn(log_2 n + 1)$

则$T(n) = \Theta(nlogn)$.

由上，原命题得证。

(d)$T(n) = \Theta(nlog(logn))$

证明如下：

令$n = 2^k$，代入递归式则有：$T(2^k) = 2^{\frac{k}{2}} T(2^{\frac{k}{2}}) + 2^k$

即$\frac{T(2^k)}{2^k} = \frac{T(2^{\frac{k}{2}})}{2^{\frac{k}{2}}} + 1$

令$S(k) = \frac{T(2^k)}{2^k}$，转化得到递归式：

$S(k) = S(\frac{k}{2}) + 1$

将递归式代入主方法公式：$S(k) = aS(k/b) + f(k)$，可知：

$a = 1, b = 2, f(k) = 1$

显然有$f(k) = 1 = \Theta(k^{log_b a}) = \Theta(k^0)$

故该递归式满足主方法的情况2，故$S(k) = \Theta(k^{log_b a} logk) = \Theta(logk)$

即$\frac{T(2^k)}{2^k} = \Theta(logk) => T(2^k) = \Theta(2^k logk)$

故$T(n) = \Theta(nlog(logn))$.

故原命题得证。

问题2

答：

对二维FFT作分析，设计如下算法：

$$
\begin{aligned}
y[k_1, k_2] &= \sum_{j_1=0}^{N-1}\sum_{j_2=0}^{N-1} exp[\frac{2\pi i(k_1 j_1 + k_2 j_2)}{N}] \cdot x[j_1, j_2] \\
&= \sum_{j_2=0}^{N-1}(\sum_{j_1=0}^{N-1} x[j_1, j_2] \times exp(\frac{2\pi i k_1 j_1}{N}) \times exp(\frac{2\pi i k_2 j_2}{N}))
\end{aligned}
$$

$\text{调用算法导论中}RECURSIVE-FFT\text{函数对括号内执行一维}FFT\text{运算，并令}z[k_1, j_2] = \sum_{j_1=0}^{N-1} x[j_1, j_2] \times exp(\frac{2\pi i k_1 j_1}{N})$

$\text{故上式} = \sum_{j_2=0}^{N-1} z[k_1, j_2] \times exp(\frac{2\pi i k_2 j_2}{N})$

$\text{再调用算法导论中}RECURSIVE-FFT\text{函数对上式执行一维}FFT\text{运算，即可算得}y[k_1, k_2]\text{结果}.$

算法实现伪代码如下：

```
1  FFT(x)
2      let z[1..N][1..N] and y[1..N][1..N] be new matrix
3      x'=transpose matrix of x
4      for i=0 to N
5          z'[i]=RECURSIVE-FFT(x'[i])
6      z=transpose matrix of z'
7      for i=0 to N
8          y[i]=RECURSIVE-FFT(z[i])
9      return y
10
```

时间复杂度分析：

根据以上设计的算法，第一轮计算$z[k_1, j_2]$需执行N次一维FFT运算(即伪代码中第4行)，第二轮计算$y[k_1, k_2]$需执行N次一维FFT运算(即伪代码中第7行)，故时间复杂度：

$T(N) = (N + N) * T_{1D\_FFT} = (N + N)NlogN = 2N^2 logN = O(N^2 logN)$

以上可证，该算法时间复杂度为$O(N^2 logN)$。

正确性分析：

利用数学归纳法证明：对于d维FFT($y \in R^{N^d}$，N是2的整数次幂)的求解皆可用**通过计算一维FFT逐渐减小维数进而计算多维FFT**的方法通过扩展以上算法实现。

①维数=1时

计算一维FFT可直接调用$RECURSIVE-FFT$算法计算，故显然以上结论正确。

②维数>1时

令维数=d且维数为d-1时满足假设条件。

则有:

$$
\begin{aligned}
&y[k_1,k_2,\ldots,k_d] \\
&=\sum_{j_1=0}^{N-1}\sum_{j_2=0}^{N-1}\cdots\sum_{j_d=0}^{N-1} exp[\frac{2\pi i(k_1j_1+k_2j_2+\ldots+k_dj_d)}{N}]\cdot x[j_1,j_2,\ldots j_d] \\
&=\sum_{j_d=0}^{N-1}(\sum_{j_1=0}^{N-1}\cdots\sum_{j_{d-1}=0}^{N-1} x[j_1,j_2,\ldots,j_d]\times exp(\frac{2\pi i(k_1j_1+\ldots+k_{d-1}j_{d-1})}{N})\times exp(\frac{2\pi ik_dj_d}{N})) \\
&\text{由于已知存在} y[k_1,\ldots,k_{d-1}]=\sum_{j_1=0}^{N-1}\cdots\sum_{j_{d-1}=0}^{N-1} x[j_1,j_2,\ldots,j_{d-1}]\times exp(\frac{2\pi i(k_1j_1+\ldots+k_{d-1}j_{d-1})}{N}) \\
&\text{故加入} j_d \text{可令} z[k_1,\ldots,k_{d-1},j_d]=\sum_{j_1=0}^{N-1}\cdots\sum_{j_{d-1}=0}^{N-1} x[j_1,j_2,\ldots,j_d]\times exp(\frac{2\pi i(k_1j_1+\ldots+k_{d-1}j_{d-1})}{N}) \\
&\text{故上式}=\sum_{j_d=0}^{N-1} z[k_1,\ldots,k_{d-1},j_d]\times exp(\frac{2\pi ik_dj_d}{N}) \\
&\text{再调用算法导论中} RECURSIVE-FFT \text{函数对上式执行一维} FFT \text{运算，即可算得} y[k_1,\ldots,k_d] \text{结果.}
\end{aligned}
$$

故可证，若维数为d-1时满足假设条件，则维数为d时也满足假设条件。

综上可证，“对于d维FFT($y \in R^{N^d}$)的求解皆可用**通过计算一维FFT逐渐减小维数进而计算多维FFT**的方法通过扩展以上算法实现”这一结论正确。

令d=2，即可证得该算法正确。

问题3

答:

(a)

设计算法伪代码如下:

```
MERGE(A, p, q, r)
   count=0    // 记录逆序对个数
   n1=q-p+1
   n2=r-q
   let L[1..n1+1] and R[1..n2+1] be new arrays
   for i=1 to n1
      L[i]=A[p+i-1]
   for j=1 to n2
      R[j]=A[q+j]
   L[n1+1]=∞
   R[n2+1]=∞
   i=1
   j=1
   for k = p to r
      if (L[i]<=R[j])
         A[k]=L[i]
         i++
      else
         A[k]=R[j]
         count+=(j+n1-k+p-1)
         j++
    return count


MERGE-SORT(A, p, r)
    if p<r
       q=⌊(p+r)/2⌋
       count=0
       count+=MERGE-SORT(A, p ,q)
       count+=MERGE-SORT(A, q+1, r)
       count+=MERGE(A, p, q, r)
```

```
32          return count
33      return 0
```

以上伪代码是由归并排序算法修改而来，原理如下：

利用归并排序的方法对数组执行排序，**MERGE**算法用于计算A[p..q]中的元素和A[q+1..r]中的元素之间存在的逆序数，**MERGE-SORT**算法用于计算A[p..r]之间存在的所有逆序数。

由于在A数组中，L数组中的元素在R数组之前，故若在比较结构中存在**L[i]>R[j]**，则必然存在逆序对，且R[j]元素在排序后的数组A[q..r]中为第k-p+1位，而在原数组A中为第n1+j位，故存在n1+j-k+p-1个逆序对，依次规律即可分治得到数组中逆序对总数。

时间复杂度证明：

显然，**MERGE**函数时间复杂度为$\Theta(N)$，则**MERGE-SORT**函数时间复杂度满足如下式子：

$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+cn & n>1\end{cases}$$

利用主方法，由于$cn=\Theta(n^{log_2 2})=\Theta(n)$，则满足主方法的第**2**种情况，故$T(n)=\Theta(n^{log_2 2}logn)=\Theta(nlogn)$。

由于数组长度为N，故时间复杂度为$\Theta(NlogN)$。

正确性证明：

循环不变式：

MERGE函数中开始第14~21行for循环的每次迭代之前，R[j]是R数组中未将“由R数组中元素与L数组中元素构成的逆序数个数”加入count的最小元素。

初始化：

第一次迭代之前，j=1且count=0，显然由R[1]与L数组中元素构成的逆序数个数尚未加入count中，且R数组中不存在比R[1]更小的元素。

保持：

在循环中判断条件时，若$L[i]>R[j]$，则$A[k]=R[j]$，并将R[j]元素与L数组中元素构成的逆序数个数加入count，而后j加1得到j'，则新一轮循环中R[j']元素与L数组中元素构成的逆序数个数未加入count且R[j']是R数组中满足该条件的最小元素；若$L[i]\leq R[j]$，则$A[k]=L[i]$，则由于该轮循环开始前满足R[j]是R数组中未将“由R数组中元素与L数组中元素构成的逆序数个数”加入count的最小元素，由于这一轮循环中j不发生变化，故下一轮循环开始之前依旧满足该条件。

终止：

终止时k=r+1，由于L、R数组中的所有元素皆已存入A数组中，故j=n2+1。算法中定义R[n2+1]=∞，即哨兵结点，故只需将R[1..n2]中的元素与L数组元素构成的逆序对加入count，而无需考虑R[n2+1]。此时满足R[n2+1]是R数组中未将“由R数组中元素与L数组中元素构成的逆序数个数”加入count的最小元素。

以上可知**MERGE**算法正确。

对**MERGE-SORT**考虑以下情况：

①N=1时

A数组仅有一个元素，不存在逆序对，算法显然正确。

②N>1时

由于**MERGE-SORT(A, p ,q)**可返回A[p..q]中所有逆序对个数，**MERGE-SORT(A, q+1, r)**可返回A[q+1..r]中所有逆序对个数，**MERGE(A, p, q, r)**可返回A[p..q]数组元素与A[q+1..r]数组元素构成的逆序对个数，以上三者相加即为A[p..r]数组中所有逆序对个数。

综上所述，**MERGE-SORT**算法正确。

(b)

设计算法伪代码如下：

```
1  MERGE(A, p, q, r, C, D)
2     n1=q-p+1
3     n2=r-q
4     let L[1..n1+1] and R[1..n2+1] be new arrays
5     for i=1 to n1
6        L[i]=A[p+i-1]
7     for j=1 to n2
8        R[j]=A[q+j]
9     L[n1+1]=∞
10    R[n2+1]=∞
11    i=1
12    j=1
13    for k = p to r
14       if (L[i]<=R[j])
15          A[k]=L[i]
16          D[B[A[k]]]+=(k-p+1-i)  //储存原数组中存在的B[L[m]]元素在前的逆序对的个数
17          i++
18       else
19          A[k]=R[j]
20          C[B[A[k]]]+=(j+n1-k+p-1);  //储存原数组中存在的B[A[k]]元素在后的逆序对的个数
21          j++
22
23
24 MERGE-SORT(A, p, r, C, D)
25    if p<r
26       q=⌊(p+r)/2⌋
27       MERGE-SORT(A, p ,q, C, D)
28       MERGE-SORT(A, q+1, r, C, D)
29       MERGE(A, p, q, r, C, D)
30
31 COUNT(A)
32    len=A.length
33    count=0
34    Let B[1..99999] and C[1..len] and D[1..len] be new arrays
35    C=D={0}
36    for i=1 to len
37       B[A[i]]=i    // 储存原数组元素在原数组中的位置
38    MERGE-SORT(A, 1, len, C, D)
39    for i=1 to len
40       count+=D[i]*(i-1-C[i])
41    return count
```

以上伪代码是由归并排序算法修改而来，原理如下：

开一个足够大的B数组，用于储存原A数组元素在原数组中的位置；开两个和A数组大小相同的数组C、D，C数组中C[i]表示原数组中存在的A[i]在后的二元逆序对的个数，D数组中D[i]表示原数组中存在的A[i]在前的二元逆序对的个数。

判断是否存在二元逆序对的方式同(a)，由于在算法18~21行中找到了A[k]元素在后的j+n1-k+p-1个二元逆序对，故找到A[k]元素在原数组中对应位置B[A[k]]，并使C[B[A[k]]]增加j+n1-k+p-1；在算法14~17行中找到了A[k]元素在前的k-p+1-i个二元逆序对，故找到A[k]元素在原数组中对应位置B[A[k]]，并使D[B[A[k]]]增加k-p+1-i。

分治算法结束后，对A数组中的每一位元素分别计算：D[i]为在原数组中A[i]右侧和A[i]构成二元逆序数的个数，如此即满足题中$j<k, A[j]>A[k]$条件；i-1为原数组i位元素左侧的元素个数，C[i]为原数组中A[i]左侧和A[i]构成二元逆序数的个数，i-1-C[i]为原数组中A[i]左侧和A[i]不构成二元逆序数的个数，如此即满足题中$i<j, A[i]<A[j]$条件。两者相乘并将每一位相乘结果相加即为所求三元组个数。

时间复杂度证明：

显然，**MERGE**函数时间复杂度为$\Theta(N)$，则**MERGE-SORT**函数时间复杂度满足如下式子：

$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+cn & n>1\end{cases}$$

利用主方法，由于$cn=\Theta(n^{log_2 2})=\Theta(n)$，则满足主方法的第**2**种情况，故$T(n)=\Theta(n^{log_2 2}logn)=\Theta(nlogn)$。**COUNT**函数中35行和38行的for循环结构时间复杂度均为$\Theta(n)$，故该函数的时间复杂度取决于**MERGE-SORT**函数，即为$\Theta(nlogn)$。

由于数组长度为N，故时间复杂度为$\Theta(NlogN)$。

正确性证明：

循环不变式：

MERGE函数中开始第14~21行for循环的每次迭代之前，L[i]是L数组中未将和R数组中元素构成二元逆序数的个数记录在D数组中的最小元素，R[j]是R数组中未将和L数组中元素构成二元逆序数的个数记录在C数组中的最小元素。

初始化：

第一次迭代之前，i=j=1，且C、D数组依旧置0，故显然满足L[1]是L数组中未将和R数组中元素构成二元逆序数的个数记录在D数组中的最小元素，R[1]是R数组中未将和L数组中元素构成二元逆序数的个数记录在C数组中的最小元素。

保持：

在循环中判断条件时，若$L[i]>R[j]$，则$A[k]=R[j]$，由于A[k]开始为A数组中的第n1+j位，排序后置于第k-p+1位，故R[j]和L数组中元素构成j+n1-k+p-1个二元逆序数并计入C数组，而后j加1得到j'，则新一轮循环中R[j']是R数组中未将和L数组中元素构成二元逆序数的个数记录在C数组中的最小元素。对于L数组，由于上一轮迭代之前满足L[i]是L数组中未将和R数组中元素构成二元逆序数的个数记录在D数组中的最小元素，迭代一轮后i不变，则依旧满足。

若$L[i]\leq R[j]$，则$A[k]=L[i]$，由于A[k]开始为A数组中的第i位，排序后置于第k-p+1位，故L[i]和R数组中元素构成k-p+1-i个二元逆序数并计入D数组，而后i加1得到i'，则新一轮循环中L[i']是L数组中未将和R数组中元素构成二元逆序数的个数记录在D数组中的最小元素。对于R数组，由于上一轮迭代之前满足R[j]是R数组中未将和L数组中元素构成二元逆序数的个数记录在C数组中的最小元素，迭代一轮后j不变，则依旧满足。

终止：

终止时k=r+1，由于L、R数组中的所有元素皆已存入A数组中，故i=n1+1，j=n2+1。算法中定义L[n1+1]=∞,R[n2+1]=∞，即哨兵结点，故无需考虑这两个结点。此时满足L[i]是L数组中未将和R数组中元素构成二元逆序数的个数记录在D数组中的最小元素，R[j]是R数组中未将和L数组中元素构成二元逆序数的个数记录在C数组中的最小元素。

由上可知，该算法正确。

问题4

答：假设班上有n位同学，分以下情况讨论：

①$n>365$时

假设班上不存在生日相同的两个人，即班上的同学一共有n种不同的生日日期。由于一年有365天，故显然假设不正确，班上存在生日相同的两人概率为**100%**。

②$n\leq 365$时

显然每一位同学的生日皆有365种可能，n位同学的生日一共有365^n种情况。

其中满足**不存在生日相同的两位同学**的情况有$365(365-1)...(365-n+1)$种，即已经有同学出生的日期不可成为其他同学的生日。

则应当满足$P=\frac{365(365-1)...(365-n+1)}{365^n}<0.5$，即不存在生日相同的两位同学的概率不超过50%。

有：

$P = \frac{365(365-1)...(365-n+1)}{365^n} = \frac{365!}{365^n \times (365-n)!} < 0.5$

由于$\frac{P_{n+1}}{P_n} = \frac{365(365-1)...(365-n)}{365^{n+1}} \div \frac{365(365-1)...(365-n+1)}{365^n} = \frac{365-n}{365} \leq 1$

故概率P随n的递增而递减.

利用二分查找法

$n = 0$时，$\frac{365!}{365^n \times (365-n)!} = 1 > 0.5$

$n = 365$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.000000 < 0.5$

$n = (0 + 365)/2 = 182$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.000000 < 0.5$

$n = (0 + 182)/2 = 91$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.000005 < 0.5$

$n = (0 + 91)/2 = 45$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.059024 < 0.5$

$n = (0 + 45)/2 = 22$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.524305 > 0.5$

$n = (22 + 45)/2 = 33$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.225028 < 0.5$

$n = (22 + 33)/2 = 27$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.373141 < 0.5$

$n = (22 + 27)/2 = 24$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.461656 < 0.5$

$n = (22 + 24)/2 = 23$时，$\frac{365!}{365^n \times (365-n)!} \approx 0.492703 < 0.5$

故$n \geq 23$时，$\frac{365!}{365^n \times (365-n)!} < 0.5$。

综上，班上至少要有23名同学，才能保证存在生日相同的两位同学的概率超过50%。

问题5

答:

(a)

假设d叉堆的数据存于数组$A[1..A.length]$中的$A[1..A.heap-size]$部分，其中$0 \leq A.heap-size \leq A.length$。

树的根结点为$A[1]$，则数组表示方式推导如下

首先分析结点i(即数组中$A[i]$表示的结点)的孩子结点在数组中的位置,显然结点i存在d个孩子结点,由于结点i之前有$i-1$个结点,故这$i-1$个结点共有$d(i-1)$个孩子结点。
这$d(i-1)$个结点加上不是任何结点的孩子结点的根结点$A[1]$,即结点i的孩子结点之前有$d(i-1)+1$个结点。由于结点i存在d个孩子结点,故第j个孩子结点在数组中位置为$d(i-1)+1+j$。

其次分析结点i的父结点在数组中的位置。令结点i的父结点为a。由以上结论可知，结点a的孩子结点范围为$[d(a-1)+2, d(a-1)+1+d]$，即i的范围为$[d(a-1)+2, da+1]$。故$a = \lfloor \frac{i-2}{d} + 1 \rfloor$。

综上，可知用数组表示d叉堆的方式如下:

```
PARENT(i)
   return ⌊(i - 2) / d + 1⌋

CHILD(i, j)     // j表示i结点的第j个孩子结点
   return d(i - 1) + 1 + j
```

(b)

根据算法导论提供的算法**MAX-HEAPIFY**、**HEAP-EXTRACT-MAX**，根据最小k叉堆的性质作修改。

实现伪代码如下:

```
EXTRACT-MIN(A)
   if A.heap-size < 1
      error "heap underflow"
   min = A[1]
   A[1] = A[A.heap-size]
```

```
    A.heap-size = A.heap.size - 1
    MIN-HEAPIFY(A, 1)
    return min

MIN-HEAPIFY(A, i)
    smallest = i
    for j = 1 to d
        if CHILD(i, j) ≤ A.heap-size and A[CHILD(i, j)] < A[i]
            if A[CHILD(i, j)] < smallest
                smallest = A[CHILD(i, j)]
    if smallest != i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

首先对函数**MIN-HEAPIFY**时间复杂度作分析：

由于该函数需迭代操作,且每一次迭代需找出**smallest**结点和一层结点之间最小的结点,故一共需迭代$O(h)$次(h为堆的高度)；且每一次迭代都需执行一次长度为**d**的循环结构,故函数**MIN-HEAPIFY**的时间复杂度$T(n)=O(dh)=O(dlog_dn)$。

对函数**EXTRACT-MIN**时间复杂度作分析：

该函数中除了时间复杂度为$O(dlog_dn)$的MIN-HEAPIFY函数外，其他操作的时间都是常数阶的，故函数**EXTRACT-MIN**的时间复杂度$T(n)=O(dlog_dn)$。

综上，算法**EXTRACT-MIN**的时间复杂度$T(n)=O(dlog_dn)$。

(c)

根据算法导论提供的算法**HEAP-INCREASE-KEY**，根据最小k叉堆的性质作修改。

实现伪代码如下：

```
1 DECREASE-KEY(A, i, k)
2     if (k > A[i])
3         error "new key is larger than current key"
4     A[i] = k
5     while i >1 and A[PARENT(i)] > A[i]
6         exchange A[i] with A[PARENT(i)]
7         i = PARENT(i)
```

对函数**HEAP-INCREASE-KEY**时间复杂度作分析：

显然，从算法第3行处理$A[i]$开始，进入$while$循环，逐渐向堆的上层移动；到最后$while$循环结束，算法从堆中的起点到终点的每一层至多停留一次，故执行时间$T(n)=O(h)=O(log_dn)$。

综上，算法**HEAP-INCREASE-KEY**的时间复杂度$T(n)=O(log_dn)$。