第一章 作业

> 智能 212 史胤隆 | 授课教师: 邢素霞

1. 某一灰度图像尺寸为 640*480，灰度级为 256，则需要多大的存储空间？若该图像为彩色图像，则又需要多大的存储空间?

由题，256 级灰度即 8 位，因此，若不考虑文件头等其余信息，至少需

$$640 \times 480 \times 8 = 2457600\text{bit} = 307200\text{B} = 300\text{KB}$$

若为彩色图像，假设存在三个通道，每个通道 8 位 (例如 RGB)，则至少需

$$640 \times 480 \times 8 \times 3 = 7372800\text{bit} = 921600\text{B} = 900\text{KB}$$

2. 查阅资料：

- **了解位图文件的文件头和位图信息部分的具体组成，分析其中哪些内容可以通过直接观察位图本身而得到。**

  网络查询“位图文件的文件头和位图信息部分一般都包含哪些信息”，得到的解释如下：

  > 位图文件通常包括文件头 (File Header) 和位图信息部分 (Bitmap Information Header)，不同类型的位图文件可能会有不同的格式，但一般情况下，位图文件的文件头包含以下一些常见的信息：
  >
  > 文件头 (File Header) 通常包括：
  >
  > | 字段 | 描述 |
  > | --- | --- |
  > | 文件类型标识符 (File Type Identifier) | 通常是 "BM"，用于标识文件为位图文件。 |
  > | 文件大小 (File Size) | 整个位图文件的大小，包括文件头和位图数据。 |
  > | 保留字段 (Reserved) | 通常为 0，保留供将来使用。 |
  > | 位图数据偏移量 (Data Offset) | 位图数据相对于文件起始位置的偏移量，即文件头的长度。 |
  >
  > 位图信息部分 (Bitmap Information Header) 通常包括：
  >
  > | 字段 | 描述 |
  > | --- | --- |
  > | 位图信息头大小 (Bitmap Info Header Size) | 指定位图信息头的大小，不同类型的位图可以有不同的信息头格式。 |
  > | 图像宽度 (Image Width) | 图像的宽度，以像素为单位。 |
  > | 图像高度 (Image Height) | 图像的高度，以像素为单位。 |
  > | 颜色平面数 (Color Planes) | 颜色平面的数量，通常为 1 或 3。 |

字段	描述
每像素位数 (Bits per Pixel)	每个像素占用的位数，决定了颜色深度。
压缩方式 (Compression Method)	指定图像的压缩方式，通常为 0 表示不压缩。
图像数据大小 (Image Data Size)	位图数据的大小，不包括文件头，以字节为单位。
水平分辨率 (Horizontal Resolution)	图像的水平分辨率，通常以像素每米为单位。
垂直分辨率 (Vertical Resolution)	图像的垂直分辨率，通常以像素每米为单位。
颜色表大小 (Color Table Size)	颜色表的大小，对于一些位图格式，包含调色板信息。
重要颜色数 (Important Colors)	对于索引色位图，表示有多少种重要的颜色，通常为 0。

通过阅读以上内容，可以通过直接观察位图本身而得到的信息包括图像宽、高度，水平、垂直分辨率等。另外，根据图像是否彩色、是否具有透明、增白、加黑等辅助通道，以及根据图像色彩的细腻程度，也可以大致猜测出图像的颜色平面数、每像素位数、压缩方式、颜色表大小等信息。

- **了解动画 GIF 文件的文件头的具体组成，能否确定其中的数据哪些是全局数据，哪些是局部数据。**

  网络查询“动画 GIF 文件的文件头具体组成”，得到的解释如下：

  > 动画 GIF 文件的文件头由以下部分组成：
  >
  > | 字段 | 描述 |
  > | --- | --- |
  > | GIF 文件签名 (GIF Signature) | 前三个字节通常是 "GIF"，用于标识文件格式。 |
  > | 版本信息 (Version) | 三个字节表示 GIF 文件的版本，通常为 "87a" 或 "89a"。 |
  > | 逻辑屏幕宽度 (Logical Screen Width) | 两个字节，指定动画的逻辑屏幕宽度（像素）。 |
  > | 逻辑屏幕高度 (Logical Screen Height) | 两个字节，指定动画的逻辑屏幕高度（像素）。 |
  > | 全局颜色表标志 (Global Color Table Flag) | 一个字节，指示是否存在全局颜色表，最高位为 1 表示存在。 |
  > | 颜色分辨率 (Color Resolution) | 一个字节，指定每个颜色表中的颜色位数减 1。 |
  > | 排序标志 (Sort Flag) | 一个字节，用于指示颜色表中的颜色是否按重要性排序，通常为 0。 |
  > | 全局颜色表大小 (Global Color Table Size) | 一个字节，指定全局颜色表中的颜色数量。 |
  > | 背景颜色索引 (Background Color Index) | 一个字节，表示背景颜色在全局颜色表中的索引。 |

> | 字段 | 描述 |
> | --- | --- |
> | 像素宽高比 (Pixel Aspect Ratio) | 一个字节，通常为 0，表示像素宽高比为 1:1。 |
>
> 这些信息组成了动画 GIF 文件的文件头，其中最重要的是逻辑屏幕宽度、逻辑屏幕高度、全局颜色表标志以及全局颜色表大小，它们定义了动画的基本属性和颜色信息。接下来的 GIF 数据块中包含了动画的帧数据和帧之间的延迟等信息。

通常，这些处于文件头的字段均为全局数据，而 GIF 数据块中的数据则为局部数据。

- **比较对应二值图像的 TIFF 头文件和对应的灰度图像的 TIFF 文件头的区别。**

  网络查找“二值图像和灰度图像的 TIFF 有什么区别”，得到的解释如下：

  > 由于两者位深度不同，二值图像属于 TIFF 中的 `1` 模式，不包含颜色配置索引，而灰度图像属于 TIFF 中的 `L` 模式，包含颜色配置索引。

  我们用经典的“莱娜图”为例，进行验证：

  我们选取一张来自互联网的标准 512*512 的莱娜图 `Lenna.png`，其大小为 462 KB (473,831 字节)
  转换为 TIFF 灰度图像并保存，其大小为 280 KB (287,392 字节)
  转换为 TIFF 二值图像并保存，其大小为 58.2 KB (59,644 字节)

  TIFF 文件通常以文件头、IFD (Image File Directory)、图像数据三部分组成。文件头的格式基本一致，大多数信息包含在 IFD 中。

  TIFF 文件的文件头包含以下信息：

  > | 字段 | 长度 | 描述 |
  > | --- | --- | --- |
  > | 字节顺序标记 (字节顺序标记) | 2 字节 | 用于标识文件的字节顺序，通常为 "II" (表示小端字节序) 或 "MM" (表示大端字节序)。 |
  > | 版本号 (Version) | 2 字节 | TIFF 文件的版本号，通常为 2A00H。 |
  > | IFD 偏移量 (IFD Offset) | 4 字节 | 第一个 IFD 的偏移量，通常为 00000008H。 |

  TIFF 文件的 IFD 通常以键值对的形式存储，每条信息以特定的数字来作为标签标识，称为一个 IFD 条目 (IFD Entry)。每个 IFD 条目的格式如下：

  > | 字段 | 长度 | 描述 |
  > | --- | --- | --- |
  > | 标签数（Entry Count） | 2 字节 | 指示 IFD 中有多少个标签。 |
  > | 标签 1（Tag List） | 大多为 12 字节 | 包含字段标识符、字段类型、字段长度和字段值或字段值的偏移量。 |
  > | 标签 2（Tag List） | 大多为 12 字节 | 包含字段标识符、字段类型、字段长度和字段值或字段值的偏移量。 |
  > | 标签 3（Tag List） | 大多为 12 字节 | 包含字段标识符、字段类型、字段长度和字段值或字段值的偏移量。 |
  > | …… | …… | …… |

字段	长度	描述
标签 n（Tag List）	大多为 12 字节	包含字段标识符、字段类型、字段长度和字段值或字段值的偏移量。
下一个 IFD 的偏移量（Next IFD Offset）	4 字节	指示下一个 IFD 的偏移量，如果没有下一个 IFD，则为 00000000H。

综上，我们使用 Python 代码来解析两个文件的文件头：

```
import struct
from pathlib import Path

PATH = Path.cwd()

def read_file_header(file_path, n=256):
    with open(file_path, "rb") as f:
        return f.read(n)

def parse_tiff_header(header_bytes):
    byte_order = header_bytes[0:2]
    if byte_order == b"II":
        byte_order = "Little Endian"
    elif byte_order == b"MM":
        byte_order = "Big Endian"
    else:
        byte_order = "Unknown"
    version = struct.unpack_from("<H", header_bytes, 2)[0]
    first_ifd_offset = struct.unpack_from("<I", header_bytes, 4)[0]
    return {
        "Byte Order": byte_order,
        "Version": version,
        "First IFD Offset": first_ifd_offset,
    }

header_lenna01 = read_file_header(PATH / "Lenna01.tiff")
header_lenna02 = read_file_header(PATH / "Lenna02.tiff")
parsed_header_lenna01 = parse_tiff_header(header_lenna01)
parsed_header_lenna02 = parse_tiff_header(header_lenna02)

parsed_header_lenna01, parsed_header_lenna02
```

```
({'Byte Order': 'Little Endian', 'Version': 42, 'First IFD Offset': 8},
 {'Byte Order': 'Little Endian', 'Version': 42, 'First IFD Offset': 8})
```

由此我们可以得知，两个文件的文件头完全一致。我们继续读取 IFD 条目：

```
def parse_ifd_entry(entry_bytes, byte_order_format):
    tag, typ, count, value_offset = struct.unpack(
        byte_order_format + "HHII", entry_bytes
    )
    return {"Tag": tag, "Type": typ, "Count": count, "Value_Offset":
value_offset}
```

```
def parse_ifd(header_bytes, offset, byte_order_format="<"):
    num_entries = struct.unpack_from(byte_order_format + "H", header_bytes,
offset)[0]
    offset += 2
    entries = []
    for _ in range(num_entries):
        entry_bytes = header_bytes[offset : offset + 12]
        entries.append(parse_ifd_entry(entry_bytes, byte_order_format))
        offset += 12
    next_ifd_offset = struct.unpack_from(byte_order_format + "I",
header_bytes, offset)[
        0
    ]
    return entries, next_ifd_offset

byte_order_format = "<"
parsed_ifd_lenna01, next_ifd_offset_lenna01 = parse_ifd(
    header_lenna01, parsed_header_lenna01["First IFD Offset"],
byte_order_format
)
parsed_ifd_lenna02, next_ifd_offset_lenna02 = parse_ifd(
    header_lenna02, parsed_header_lenna02["First IFD Offset"],
byte_order_format
)

parsed_ifd_lenna01, next_ifd_offset_lenna01, parsed_ifd_lenna02,
next_ifd_offset_lenna02
```

```
([{'Tag': 254, 'Type': 4, 'Count': 1, 'Value_Offset': 0},
  {'Tag': 256, 'Type': 3, 'Count': 1, 'Value_Offset': 512},
  {'Tag': 257, 'Type': 3, 'Count': 1, 'Value_Offset': 512},
  {'Tag': 258, 'Type': 3, 'Count': 1, 'Value_Offset': 8},
  {'Tag': 259, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
  {'Tag': 262, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
  {'Tag': 273, 'Type': 4, 'Count': 1, 'Value_Offset': 25198},
  {'Tag': 274, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
  {'Tag': 277, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
  {'Tag': 278, 'Type': 3, 'Count': 1, 'Value_Offset': 512},
  {'Tag': 279, 'Type': 4, 'Count': 1, 'Value_Offset': 262144},
  {'Tag': 282, 'Type': 5, 'Count': 1, 'Value_Offset': 254},
  {'Tag': 283, 'Type': 5, 'Count': 1, 'Value_Offset': 262},
  {'Tag': 296, 'Type': 3, 'Count': 1, 'Value_Offset': 2},
  {'Tag': 305, 'Type': 2, 'Count': 31, 'Value_Offset': 270},
  {'Tag': 306, 'Type': 2, 'Count': 20, 'Value_Offset': 302},
  {'Tag': 700, 'Type': 1, 'Count': 15083, 'Value_Offset': 322},
  {'Tag': 34377, 'Type': 1, 'Count': 8880, 'Value_Offset': 15406},
  {'Tag': 34665, 'Type': 4, 'Count': 1, 'Value_Offset': 287344},
  {'Tag': 34675, 'Type': 7, 'Count': 912, 'Value_Offset': 24286}],
 0,
 [{'Tag': 254, 'Type': 4, 'Count': 1, 'Value_Offset': 0},
  {'Tag': 256, 'Type': 3, 'Count': 1, 'Value_Offset': 512},
  {'Tag': 257, 'Type': 3, 'Count': 1, 'Value_Offset': 512},
  {'Tag': 258, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
  {'Tag': 259, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
  {'Tag': 262, 'Type': 3, 'Count': 1, 'Value_Offset': 0},
```

```
 {'Tag': 273, 'Type': 4, 'Count': 1, 'Value_Offset': 27876},
 {'Tag': 274, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
 {'Tag': 277, 'Type': 3, 'Count': 1, 'Value_Offset': 1},
 {'Tag': 278, 'Type': 3, 'Count': 1, 'Value_Offset': 512},
 {'Tag': 279, 'Type': 4, 'Count': 1, 'Value_Offset': 32768},
 {'Tag': 282, 'Type': 5, 'Count': 1, 'Value_Offset': 254},
 {'Tag': 283, 'Type': 5, 'Count': 1, 'Value_Offset': 262},
 {'Tag': 296, 'Type': 3, 'Count': 1, 'Value_Offset': 2},
 {'Tag': 305, 'Type': 2, 'Count': 31, 'Value_Offset': 270},
 {'Tag': 306, 'Type': 2, 'Count': 20, 'Value_Offset': 302},
 {'Tag': 700, 'Type': 1, 'Count': 15873, 'Value_Offset': 322},
 {'Tag': 33723, 'Type': 7, 'Count': 15, 'Value_Offset': 16196},
 {'Tag': 34377, 'Type': 1, 'Count': 11664, 'Value_Offset': 16212},
 {'Tag': 34665, 'Type': 4, 'Count': 1, 'Value_Offset': 60644}],
0)
```

我们发现两者存在差异。因此，我们得出结论：**二值图像和灰度图像的 TIFF 文件头完全一致**，其包括图像模式、色彩索引等**差别均在 IFD 中。**

3. 用看图软件（如附件中的画图，Imaging，Photoshop 等），打开一幅.jpeg 格式的图像，然后另存为 bmp 和 png 格式，然后与源文件比较大小。

依然以“莱娜图”为例：

我们首先将 png 格式、大小为 462 KB (473,831 字节) 的“莱娜图”转存为 jpeg 作为“原图”

“原图”大小为 67.1 KB (68,800 字节)
转存为 bmp 格式，大小为 1.00 MB (1,048,630 字节)
转存为 png 格式，大小为 463 KB (474,628 字节)

经查阅，我们得知，jpeg 格式是有损压缩，而 bmp 和 png 格式是无损格式
且两者均包含 RGBA 四个通道，而 jpeg 仅包含 RGB 三个通道
因此在本次实验中两个图像文件均变得更大

bmp 又叫做设备无关位图，属于不压缩的位图格式，因此文件恰好为

$$512 \times 512 \times 8 \times 4 = 8388608\text{bit} = 1048576\text{B} \approx 1.00\text{MB}$$

而 png 由于存在压缩，因此图像文件小于 bmp 格式，但在本次实验中仍然比 jpeg 格式大。

当然，并非因为 jpeg 文件属于有损压缩，它就一定比对应的 png 文件小。具体证明详见下一题。

4. 打开一幅分辨率为 1024*1024 的图像，用软件更改为 800*800 的图像，比较文件修改前后的文件大小。

事实上，调整分辨率有两种常见的方式：缩放或裁切。

虽然缩小图片代表着信息量的降低，但由于文件格式和缩放算法的不同，在极少数情况下，缩小图片的文件大小可能会变大。
对于 png 图片，由于使用了色彩索引的方式，缩小图片可能使得色彩索引的数量减少，一定使像素点的数量减少，因此文件大小一般都会减小；
对于 jpeg 图片，他采用了每 8*8 个像素点为一个数据块的方式进行压缩，因此在极端情况下，文件大小反而会变大。

我们实验如下：

制作一张由随机色块生成的 1024*1024 的图片，从左上角起，每个色块的大小均为 8*8。

```
from PIL import Image, ImageDraw
import random

img_size = 1024
block_size = 8
img = Image.new("RGB", (img_size, img_size), "white")
draw = ImageDraw.Draw(img)
for y in range(0, img_size, block_size):
    for x in range(0, img_size, block_size):
        color = (random.randint(0, 255), random.randint(0, 255),
random.randint(0, 255))
        draw.rectangle([x, y, x + block_size, y + block_size], fill=color)

img.save("random_blocks.png")
```

输出的 png 文件，大小为 80.8 KB (82,802 字节)
将该图片通过 PhotoShop 转换为 jpeg 格式，调整至不出现杂色并保存，大小为 120 KB (123,333 字节)
换用更高质量保存，我们观察到文件大小不再变大，因为文件信息量已经完全。

当然，此时 jpeg 文件比 png 文件大出不少，这是因为 png 文件采用了颜色索引，更擅长处理带有大块纯色的图片。
这也证明了**并非因为 jpeg 文件属于有损压缩，它就一定比对应的 png 文件小。**

再次用 PhotoShop 打开，缩放为 800*800，我们发现，无论如何调整，杂色均无法再得到消除：
保存为标准的质量，大小为 838 KB (858,378 字节)；
即使保存为最低质量，大小仍高达 175 KB (179,675 字节)；
若保存为最高质量，大小甚至高达 1.31 MB (1,383,622 字节)。
经缩小后，**文件大小反而显著增大。**

出现以上现象的原因是，由于 jpeg 采用了每 8*8 个像素点为一个块，并以波形优化的形式进行压缩，因此 8*8 色块恰好是最佳的压缩单位，而缩放后的图片打破了这种规律，因此无法再进行压缩，反而使得文件大小变大。

综上，我们再次确定了我们的结论：缩小图片分辨率大小虽然减少了信息量，在大多数情况下文件大小也会不同程度的变小，但基于不同文件格式的特性，**在极端情况下，缩小分辨率后文件大小可能会变大。**