تابع `draw\_setup\_screen` شامل یک سری عملیات گرافیکی و یک حلقه‌ی بی‌نهایت (تا زمانی که `waiting` برابر با `False` نشود) است. دعوت از این تابع ممکن است زمان قابل توجهی را اشغال کند، زیرا این تابع تا زمانی که یکی از کلیدها فشرده نشده است در حلقه بی‌نهایت باقی می‌ماند. از این رو، زمان اجرای تابع به طور مستقیم به زمانی که کاربر نقشه را تنظیم می‌کند (با فشردن کلیدها) مرتبط است.

زمان اجرای این تابع به موارد زیر بستگی دارد:

1. \*\*پردازش‌های گرافیکی:\*\*

- `self.screen.fill((255, 255, 255))`: پر کردن صفحه با رنگ سفید.

- `manual\_text = self.font.render("M for Manual Input|R for Random Grid", True, (0, 0, 0))`: ایجاد متن گرافیکی با پویانه برای نمایش روی صفحه.

- `manual\_rect = manual\_text.get\_rect(center=(self.width // 2, self.height // 2))`: محاسبه موقعیت مرکز متن.

- `self.screen.blit(manual\_text, manual\_rect)`: نمایش متن روی صفحه.

- `pygame.display.flip()`: به‌روزرسانی نمایش صفحه.

2. \*\*حلقه انتظار و ورودی:\*\*

- `while waiting:`: حلقه بی‌نهایت تا زمانی که `waiting` برابر با `False` نشود.

- `for event in pygame.event.get()`: دریافت و پردازش وقایع.

- `if event.type == pygame.QUIT:`: اگر کاربر پنجره را ببندد، بازی خاتمه می‌یابد.

- `elif event.type == pygame.KEYDOWN:`: اگر یک کلید فشرده شود، وارد شاخه‌های مختلف می‌شود.

- `if event.key == pygame.K\_m:`: اگر کلید "M" فشرده شود، `manual\_input` به `True` تغییر می‌یابد و حلقه خاتمه می‌یابد.

- `elif event.key == pygame.K\_r:`: اگر کلید "R" فشرده شود، `manual\_input` به `False` تغییر می‌یابد، یک نقشه تصادفی ایجاد می‌شود و حلقه خاتمه می‌یابد.

در کل، زمان اجرای این تابع به میزان وقوع و پردازش وقایع توسط حلقه وابسته است. اگر کاربر سریعاً واکنش نشان دهد (کلید فشرده شود)، زمان اجرای تابع نیز کاهش خواهد یافت. اما اگر کاربر برای مدت زمان طولانی اقدام به فشردن کلید نکند، تابع ممکن است زمان زیادی اجرا شود.

تابع `create\_random\_grid` یک صفحه ابتدایی برای بازی Sudoku با مقادیر تصادفی ایجاد می‌کند. برای تحلیل زمانی این کد، به هر یک از بخش‌های اصلی تابع نگاه می‌کنیم:

1. \*\*پر کردن صفحه با صفر:\*\*

- `for i in range(9):`: یک حلقه از 0 تا 8 برای حلقه خارجی.

- `for j in range(9):`: یک حلقه دیگر از 0 تا 8 برای حلقه داخلی.

- `self.board[i][j] = 0`: یک اختصاص مستقیم که در O(1) انجام می‌شود.

بنابراین، زمان اجرای این بخش از تابع O(81) یا به صورت مستقیم O(1) خواهد بود.

2. \*\*تولید یک نقشه تصادفی:\*\*

- `for \_ in range(20):`: یک حلقه 20 بار برای انتخاب 20 خانه تصادفی.

- `row, col = random.randint(0, 8), random.randint(0, 8)`: دو عدد تصادفی برای شناسایی مکان خانه.

- `num = random.randint(1, 9)`: یک عدد تصادفی برای قرار دادن در خانه.

- `while not self.is\_valid(num, (row, col)): ...`: یک حلقه while که تا زمانی که عدد تصادفی در مکان صحیح قرار نگرفته، عدد و مکان جدیدی تولید می‌کند.

- `self.board[row][col] = num`: قرار دادن عدد تصادفی در مکان صحیح.

اگر این حلقه 20 بار اجرا شود، زمان اجرای این بخش از تابع به تعداد تکرارها یا O(20) خواهد بود.

بنابراین، زمان اجرای کل تابع می‌تواند به طور تقریبی به صورت O(1) + O(20) = O(20) تخمین زده شود.

تابع `draw\_board` برای رسم صفحه بازی Sudoku استفاده می‌شود. برای تحلیل زمانی این کد، به هر یک از بخش‌های اصلی تابع نگاه می‌کنیم:

1. \*\*حلقه‌های تو در تو برای پیمایش صفحه:\*\*

- `for i in range(9):`: یک حلقه از 0 تا 8 برای پیمایش سطرها.

- `for j in range(9):`: یک حلقه دیگر از 0 تا 8 برای پیمایش ستون‌ها.

- `rect = pygame.Rect(j \* cell\_size, i \* cell\_size, cell\_size, cell\_size)`: ایجاد یک مستطیل (rect) برای هر خانه با اندازه مناسب.

- `pygame.draw.rect(self.screen, (255, 255, 255), rect)`: رسم مستطیل با رنگ سفید در صفحه.

- `pygame.draw.rect(self.screen, (0, 0, 0), rect, 2)`: رسم مستطیل با حاشیه ضخیم و رنگ سیاه.

- `if self.board[i][j] != 0: ...`: اگر مقدار خانه متفاوت از صفر باشد:

- `num\_text = self.font.render(str(self.board[i][j]), True, (0, 0, 0))`: تولید متن عدد با فونت.

- `num\_rect = num\_text.get\_rect(center=rect.center)`: تعیین مکان متن در مرکز خانه.

- `self.screen.blit(num\_text, num\_rect)`: نمایش متن عدد در مکان مشخص شده.

- `if (i, j) == self.selected: ...`: اگر خانه انتخاب شده باشد:

- `pygame.draw.rect(self.screen, (200, 200, 0), rect, 5)`: رسم یک مستطیل زرد دیگر به عنوان حاشیه خانه انتخاب شده.

بنابراین، زمان اجرای کد در این تابع به طور تقریبی به صورت O(81) یا O(1) خواهد بود.

در کل، تحلیل زمانی تابع `draw\_board` به نظر می‌رسد که از پیچیدگی زمانی پایینی برخوردار باشد و به عنوان O(1) تخمین زده می‌شود.

تابع `solve\_sudoku` به نظر می‌آید که از یک تابع بازگشتی به نام `solve` استفاده می‌کند تا یک صفحه Sudoku را حل کند. برای تحلیل زمانی این کد، نگاهی به تابع `solve` می‌اندازیم:

1. \*\*تنظیم مجدد تعداد تلاش‌ها:\*\*

- `self.attempts = 0`: یک عملیات ساده اختصاص مستقیم که در O(1) انجام می‌شود.

2. \*\*فراخوانی تابع `solve`:\*\*

- `if self.solve():`: فراخوانی تابع بازگشتی `solve`.

- اگر تابع `solve` با موفقیت اجرا شود:

- `self.print\_board()`: نمایش صفحه حل‌شده.

- اگر تابع `solve` با موفقیت اجرا نشود:

- `print("No solution exists.")`: یک پیام خطا چاپ می‌شود.

\*\*تحلیل زمانی تابع `solve`:\*\*

- `self.find\_empty()`: یک جستجوی خالی در صفحه انجام می‌دهد که در بدترین حالت به اندازه O(81) خواهد بود (تعداد کل خانه‌ها در یک صفحه Sudoku).

- حلقه `for num in range(1, 10)`: یک حلقه از 1 تا 9 برای امتحان اعداد مختلف.

- `self.is\_valid(num, (row, col))`: یک تابع دیگر را فراخوانی می‌کند که در بدترین حالت به اندازه O(81) خواهد بود.

- `self.solve()`: یک فراخوانی بازگشتی که در صورت موفقیت به تعداد خالی‌ها ادامه خواهد داد.

- `self.board[row][col] = 0`: یک عملیات اختصاص مستقیم که معمولاً در O(1) انجام می‌شود.

به طور کلی، تابع `solve` در بدترین حالت ممکن است به تعداد خانه‌های خالی در صفحه Sudoku اجرا شود، که در موارد عملی اکثراً به اندازه O(81) خواهد بود.

در نتیجه، تحلیل زمانی کد `solve\_sudoku` می‌تواند به عنوان O(81) یا به صورت مستقیم O(1) تقریب زده شود.

تابع `solve` به نظر می‌رسد که از یک الگوریتم بازگشتی برای حل مسئله Sudoku استفاده می‌کند. بیایید تحلیل زمانی این تابع را انجام دهیم:

1. `self.find\_empty()`: یک جستجوی خانه خالی انجام می‌دهد. این تابع در بدترین حالت به تعداد کل خانه‌ها، یعنی O(81) اجرا می‌شود.

2. حلقه `for num in range(1, 10)`: یک حلقه از 1 تا 9 برای امتحان اعداد مختلف. این حلقه در بدترین حالت به تعداد اعداد، یعنی O(9) اجرا می‌شود.

3. `self.is\_valid(num, (row, col))`: یک تابع دیگر را فراخوانی می‌کند که در بدترین حالت به تعداد کل خانه‌ها، یعنی O(81) اجرا می‌شود.

4. `self.solve()`: یک فراخوانی بازگشتی که در صورت موفقیت به تعداد خالی‌ها ادامه خواهد داد. تعداد خالی‌ها ممکن است به اندازه O(81) باشد.

بنابراین، تعداد کل عملیات اجرا شده در حلقه `for num in range(1, 10)` و تعداد فراخوانی‌های بازگشتی ممکن است به تعداد خالی‌ها، یعنی O(81) باشد.

در نتیجه، تحلیل زمانی این کد به صورت تقریبی می‌تواند به عنوان O(81 \* 81) یا به صورت کلی O(n^2) در نظر گرفته شود که n تعداد خانه‌های خالی در صفحه Sudoku است. اما به دلیل ثابت بودن ابعاد (9x9) صفحه Sudoku، معمولاً به عنوان O(1) در نظر گرفته می‌شود.

تابع `find\_empty` به دنبال یک خانه خالی در صفحه Sudoku می‌گردد. برای تحلیل زمانی این کد، نگاهی به حلقه‌های تو در تو داخلی آن می‌اندازیم:

1. `for i in range(9):`: حلقه بیرونی از 0 تا 8 برای پیمایش سطرها.

- `for j in range(9):`: حلقه داخلی از 0 تا 8 برای پیمایش ستون‌ها.

- `if self.board[i][j] == 0: return i, j`: اگر خانه خالی باشد، این تابع مقدار `(i, j)` را برمی‌گرداند و عملیات به پایان می‌رسد.

با توجه به اینکه این تابع هر خانه از صفحه را یک‌بار از سمت چپ به راست و از بالا به پایین پیمایش می‌کند، می‌توان تعداد کل عملیات اجرایی تابع را به تعداد کل خانه‌ها در یک صفحه Sudoku، یعنی O(81) تقریب زد.

بنابراین، تحلیل زمانی این کد به عنوان O(81) یا به صورت کلی O(n) در نظر گرفته می‌شود که n تعداد کل خانه‌ها در صفحه Sudoku است. اما به دلیل ثابت بودن ابعاد (9x9) صفحه Sudoku، معمولاً به عنوان O(1) در نظر گرفته می‌شود.

تابع `is\_valid` بررسی می‌کند که آیا یک عدد مشخص در یک موقعیت خاص در صفحه Sudoku قابل قرار گرفتن است یا نه. برای تحلیل زمانی این کد، بیایید نگاهی به قسمت‌های مختلف تابع بندازیم:

1. \*\*بررسی سطر و ستون:\*\*

- `if num in self.board[pos[0]]`: بررسی وجود `num` در سطر مورد نظر. در بدترین حالت، این اجرایی است که به اندازه O(9) اجرا می‌شود.

- `if num in [self.board[i][pos[1]] for i in range(9)]`: بررسی وجود `num` در ستون مورد نظر. در بدترین حالت، این اجرایی است که به اندازه O(9) اجرا می‌شود.

2. \*\*بررسی بلوک 3x3:\*\*

- `start\_row, start\_col = 3 \* (pos[0] // 3), 3 \* (pos[1] // 3)`: تعیین ابتدای بلوک 3x3 متناظر با موقعیت مورد نظر.

- حلقه‌های داخلی `for i in range(start\_row, start\_row + 3)` و `for j in range(start\_col, start\_col + 3)`: بررسی وجود `num` در بلوک 3x3. در بدترین حالت، این اجرایی است که به اندازه O(9) اجرا می‌شود.

3. \*\*جمع‌بندی:\*\*

- تعداد کل اجراها از O(9) در سطرها، O(9) در ستون‌ها و O(9) در بلوک‌ها می‌شود.

- بنابراین، تحلیل زمانی کل تابع به صورت O(9) در نظر گرفته می‌شود.

در نتیجه، تحلیل زمانی کد `is\_valid` به صورت تقریبی O(9) یا به صورت کلی O(1) می‌باشد. تعداد ثابت اعداد در یک صفحه Sudoku (9x9) توجیه می‌کند که زمان اجرای این تابع ثابت باشد.

تابع `print\_board` صرفاً یک حلقه `for` را بر روی لیست `self.board` اجرا می‌کند و هر ردیف را چاپ می‌کند. اگر تعداد سطرها را به عنوان n در نظر بگیریم (که در صورت یک صفحه Sudoku، n برابر با 9 است)، تحلیل زمانی این کد به صورت زیر است:

1. `for row in self.board:`: حلقه از n بار (تعداد سطرها) اجرا می‌شود.

- `print(row)`: اجرای یک دستور چاپ، که به صورت ثابت O(1) زمان می‌برد.

بنابراین، تحلیل زمانی کل تابع معادل با O(n) خواهد بود. اما اگر به تعداد کل خانه‌ها در صفحه (9x9 در صورت Sudoku) نگاه کنیم، می‌توان این تحلیل زمانی را به صورت کلی O(n^2) در نظر گرفت. این امر به دلیل ثابت بودن تعداد سطرها (n) و ستون‌ها (نیز n) در یک صفحه Sudoku است.

تابع `draw\_grid` به نظر می‌آید که خطوط مربوط به شبکه‌ی Sudoku را روی صفحه نقاشی کند. بیایید تحلیل زمانی این کد را انجام دهیم:

1. `cell\_size = self.width // 9`: یک عملیات مشترک که به صورت ثابت O(1) اجرا می‌شود.

2. `for i in range(10)`: یک حلقه از 0 تا 9 برای ایجاد خطوط شبکه.

- `pygame.draw.line(self.screen, (0, 0, 0), (i \* cell\_size, 0), (i \* cell\_size, self.height), 2)`: ایجاد خطوط افقی. این عملیات به صورت ثابت O(10) اجرا می‌شود.

- `pygame.draw.line(self.screen, (0, 0, 0), (0, i \* cell\_size), (self.width, i \* cell\_size), 2)`: ایجاد خطوط عمودی. این عملیات نیز به صورت ثابت O(10) اجرا می‌شود.

به طور کلی، تحلیل زمانی این کد به صورت O(10) خواهد بود. اما اگر به تعداد کل خانه‌ها در صفحه (9x9 در صورت Sudoku) نگاه کنیم، می‌توان تحلیل زمانی را به صورت کلی O(n) در نظر گرفت که n تعداد کل خانه‌هاست. اما به دلیل ثابت بودن ابعاد (9x9) صفحه Sudoku، معمولاً به عنوان O(1) در نظر گرفته می‌شود.

کد `get\_manual\_input` نمایش صفحه بازی Sudoku را ایجاد کرده و به کاربر این امکان را می‌دهد که اعداد را به صورت دستی وارد کند. بیایید تحلیل زمانی این کد را انجام دهیم:

1. \*\*حلقه اصلی `while inputting`:\*\*

- `for event in pygame.event.get()`: یک حلقه برای گرفتن و پردازش ورودی‌ها (رخدادها)، که در بدترین حالت به اندازه O(k) اجرا می‌شود که k تعداد رخدادها است.

- `if event.type == pygame.QUIT:`: یک شرط برای بررسی بسته شدن پنجره بازی، که به صورت O(1) اجرا می‌شود.

- `elif event.type == pygame.MOUSEBUTTONDOWN:`: یک شرط برای بررسی کلیک موس، که به صورت O(1) اجرا می‌شود.

- محاسبه مختصات سلول انتخاب شده: این نیز به صورت O(1) اجرا می‌شود.

- `elif event.type == pygame.KEYDOWN:`: یک شرط برای بررسی کلیدی که فشرده شده است، که به صورت O(1) اجرا می‌شود.

- شرط بررسی اینکه آیا یک سلول انتخاب شده است و کلید فشرده شده یکی از اعداد 1 تا 9 می‌باشد.

2. \*\*بررسی شرط اتمام `inputting = False`:\*\*

- در صورتی که شرط بررسی شود، حلقه `while` به اتمام می‌رسد.

3. \*\*پر کردن صفحه و نمایش تغییرات:\*\*

- `self.screen.fill((255, 255, 255))`: پر کردن صفحه با رنگ سفید، که به صورت O(1) اجرا می‌شود.

- `self.draw\_grid()`: نقاشی شبکه Sudoku، که به صورت O(1) اجرا می‌شود.

- `self.draw\_board()`: نقاشی اعداد در صفحه، که به صورت O(n^2) اجرا می‌شود (n تعداد خانه‌ها در یک صفحه Sudoku).

- `pygame.display.flip()`: نمایش تغییرات روی صفحه، که به صورت O(1) اجرا می‌شود.

- `self.clock.tick(60)`: محدود کردن تعداد فریم در ثانیه (FPS) به 60، که به صورت O(1) اجرا می‌شود.

به طور کلی، تحلیل زمانی این کد به صورت تقریبی O(k + n^2) خواهد بود، که n تعداد خانه‌ها و k تعداد رخدادهاست. اما به دلیل ثابت بودن تعداد خانه‌ها در صفحه Sudoku (9x9)، معمولاً به عنوان O(k) یا به صورت کلی O(1) در نظر گرفته می‌شود.

تابع `draw\_attempts\_info` به نظر می‌آید که اطلاعات تعداد تلاش‌ها را روی صفحه نقاشی کند. برای تحلیل زمانی این کد، نگاهی به عملیات انجام شده در این تابع می‌اندازیم:

1. `attempts\_text = self.font.render("Attempts: {}".format(self.attempts), True, (255, 0, 0))`: ایجاد یک متن با استفاده از فونت. این عملیات به صورت ثابت O(1) اجرا می‌شود.

2. `attempts\_rect = attempts\_text.get\_rect(center=(self.width // 2, self.height - 20))`: محاسبه مستطیل محاطی متن بر اساس مختصات مرکزی. این عملیات نیز به صورت ثابت O(1) اجرا می‌شود.

3. `self.screen.blit(attempts\_text, attempts\_rect)`: اعمال متن به صفحه. این نیز به صورت ثابت O(1) اجرا می‌شود.

به طور کلی، تحلیل زمانی این کد به صورت ثابت O(1) خواهد بود. چرا که تمام عملیات انجام شده در تابع به صورت ثابت و مستقل از تعداد خانه‌ها یا داده‌های مربوط به صفحه Sudoku است.

تابع `run` اجرا‌ی اصلی برنامه است و بر اساس شرایط مختلف ورودی و رخدادها، عملیات مختلفی را انجام می‌دهد. بیایید تحلیل زمانی این کد را انجام دهیم:

1. \*\*`self.draw\_setup\_screen()`\*\*: اجرای تابع `draw\_setup\_screen` که عملیات ترسیم صفحه اولیه بازی را انجام می‌دهد. تحلیل زمانی این تابع به صورت ثابت O(1) است.

2. \*\*`if self.manual\_input: self.get\_manual\_input()`\*\*: اگر حالت ورود دستی فعال باشد (`manual\_input=True`)، تابع `get\_manual\_input` فراخوانی می‌شود. تحلیل زمانی این تابع به صورت O(k + n^2) است، که در آن k تعداد رخدادها و n تعداد خانه‌ها در یک صفحه Sudoku است.

3. \*\*حلقه اصلی `while True`\*\*:

- `for event in pygame.event.get()`: حلقه برای گرفتن و پردازش ورودی‌ها (رخدادها)، که در بدترین حالت به اندازه O(k) اجرا می‌شود که k تعداد رخدادها است.

- `if event.type == pygame.QUIT:`: یک شرط برای بررسی بسته شدن پنجره بازی، که به صورت O(1) اجرا می‌شود.

- `elif event.type == pygame.MOUSEBUTTONDOWN:`: یک شرط برای بررسی کلیک موس، که به صورت O(1) اجرا می‌شود.

- محاسبه مختصات سلول انتخاب شده: این نیز به صورت O(1) اجرا می‌شود.

- `elif event.type == pygame.KEYDOWN:`: یک شرط برای بررسی کلیدی که فشرده شده است، که به صورت O(1) اجرا می‌شود.

- شرط بررسی اینکه آیا یک سلول انتخاب شده است و کلید فشرده شده یکی از اعداد 1 تا 9 یا Enter می‌باشد.

- `self.screen.fill((255, 255, 255))`: پر کردن صفحه با رنگ سفید، که به صورت ثابت O(1) اجرا می‌شود.

- `self.draw\_grid()`: نقاشی شبکه Sudoku، که به صورت ثابت O(1) اجرا می‌شود.

- `self.draw\_board()`: نقاشی اعداد در صفحه، که به صورت O(n^2) اجرا می‌شود (n تعداد خانه‌ها در یک صفحه Sudoku).

- `self.draw\_attempts\_info()`: نقاشی اطلاعات تعداد تلاش‌ها، که به صورت ثابت O(1) اجرا می‌شود.

- `pygame.display.flip()`: نمایش تغییرات روی صفحه، که به صورت ثابت O(1) اجرا می‌شود.

- `self.clock.tick(60)`: محدود کردن تعداد فریم در ثانیه (FPS) به 60، که به صورت ثابت O(1) اجرا می‌شود.

به طور کلی، تحلیل زمانی این کد به صورت تقریبی O(k + n^2) می‌باشد. اما به دلیل ثابت بودن ابعاد (9x9) صفحه Sudoku، معمولاً به عنوان O(k) در نظر گرفته می‌شود.