



گزارش کار آزمایش اول آزمایشگاه مهندسی نرم افزار

دانشکده مهندسی کامپیوتر

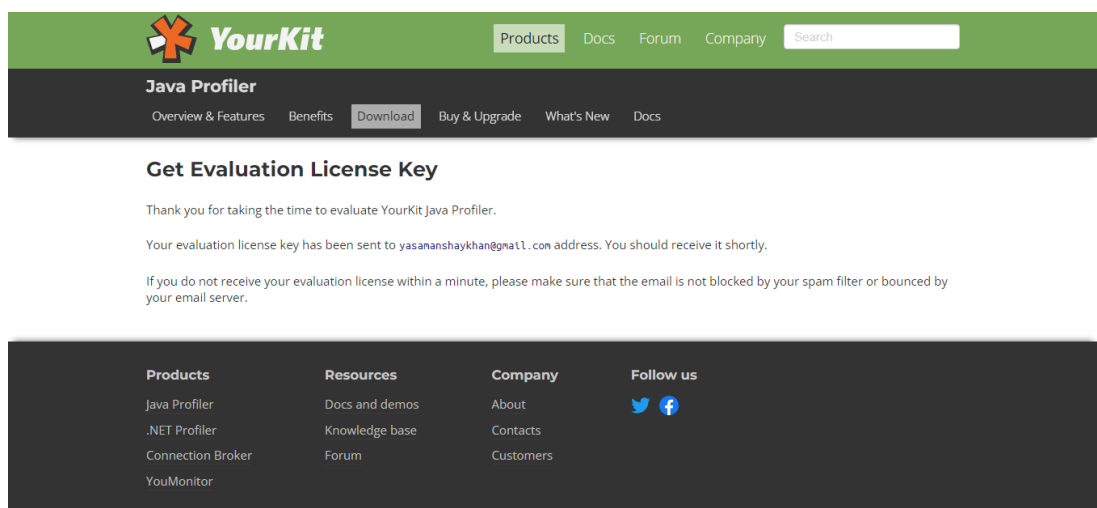
لینک گیت‌هاب: https://github.com/yasmansh/SE_LAB (دسترسی به دستیار آموزشی داده شده است)

اعضای گروه:

یاسمن شیخان - 97101915

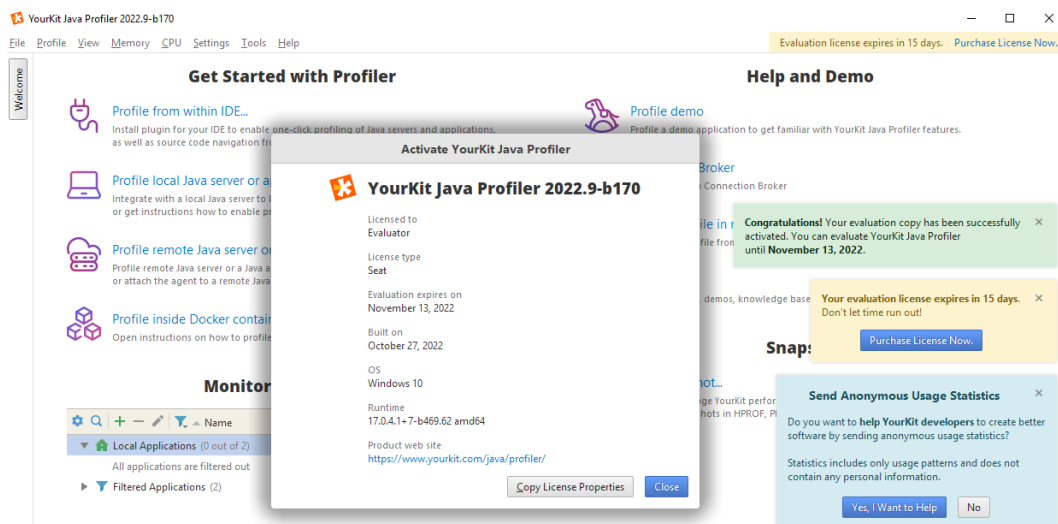
امیرحسین علی محمدی - 97110166

ابتدا به بررسی تصاویری از مراحل نصب برنامه **YourKit-JavaProfiler** می پردازیم. با مراجعه به وبسایت <https://www.yourkit.com/download>، آخرین نسخه مطابق با سیستم عامل (در اینجا ویندوز) را دانلود کرده و سپس آن را اجرا می کنیم. برای دریافت مجوز 15 روزه ایمیل خود را وارد و **lisence key** را دریافت می کنیم.



شکل ۱ - دریافت کلید مجوز ۱۵ روزه از طریق ایمیل

سپس کلید ارسال شده را در برنامه وارد کرده و مجوز استفاده از آن فعال می شود.

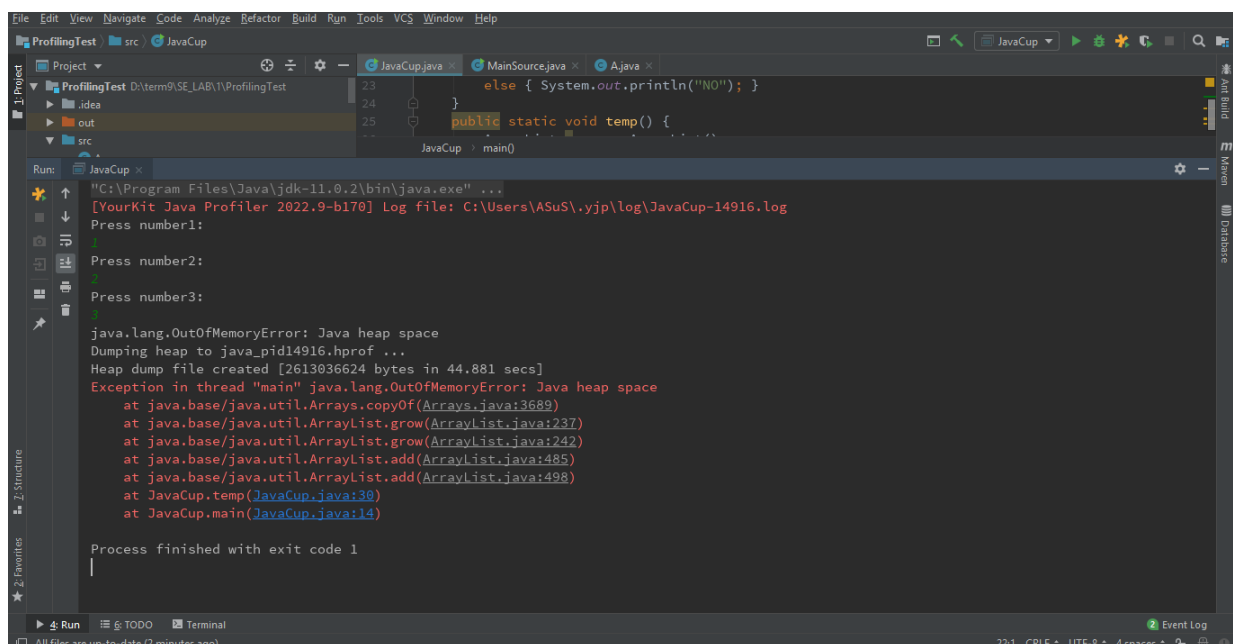


شکل ۲ - تصویری از محیط برنامه پس از فعال سازی

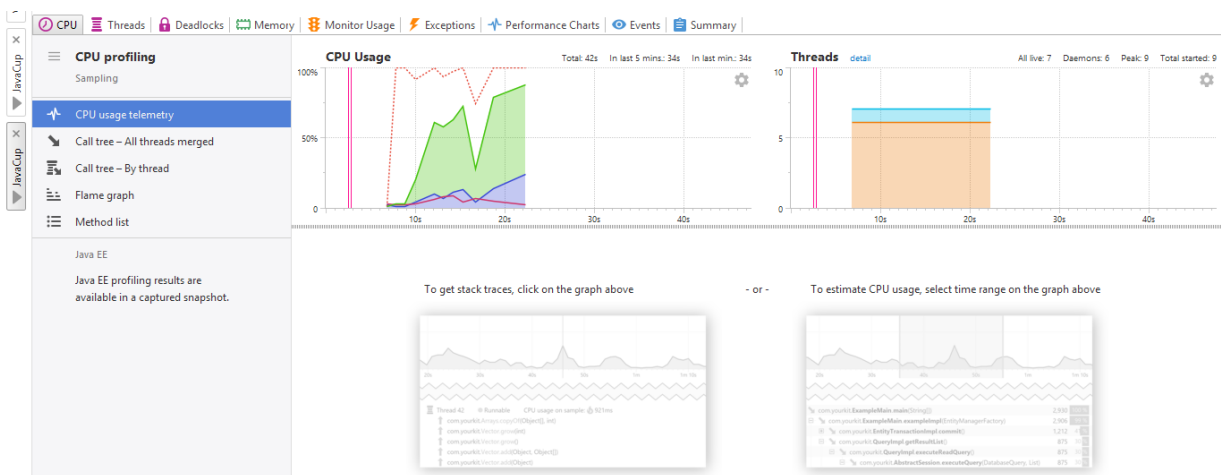
در ادامه به بررسی هر دو بخش آزمایش می پردازیم.

بخش اول.

در پروژه **ProfilingTest**، عملیات **Profiling** را بر روی کلاس **JavaCup** اجرا کرده و با استفاده از **YourKit** به بررسی مصرف منابع، تحلیل حافظه و **CPU** و **Thread** ها خواهیم پرداخت.



شکل ۳ - خروجی برنامه بعد از اجرای عملیات *profiling* (قبل از بهبود)



شکل ۴ - نمودار استفاده از پردازنده (قبل از بهبود)

طبق نمودار مربوط به thread ها و زمان اجرای توابع مشاهده می شود 91 درصد زمان اجرایی توسط تابع temp مصرف می شود.

Call Tree	Time (ms)	Samples
<All threads>	4,328	750
JavaCup.java:14 JavaCup.temp()	4,171	342
JavaCup.java:29	3,921	168
JavaCup.java:30 java.util.ArrayList.add(Object)	2,718	147
JavaCup.java:30 java.lang.Integer.valueOf(int)	1,031	10
JavaCup.java:30	156	10
JavaCup.java:30	15	1
JavaCup.java:7 java.util.Scanner.<init>(InputStream)	156	8
JavaCup.java:7 java.util.Scanner.<clinit>()	78	4
JavaCup.java:9 java.util.Scanner.nextInt()	15	66
JavaCup.java:11 java.util.Scanner.nextInt()	0	59
JavaCup.java:13 java.util.Scanner.nextInt()	0	37
com.intellij.execution.application.AppMainV2\$1.run()	125	375
sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, String)	31	33

شکل ۵ - نمودار درختی مربوط به thread ها (قبل از بهبود)

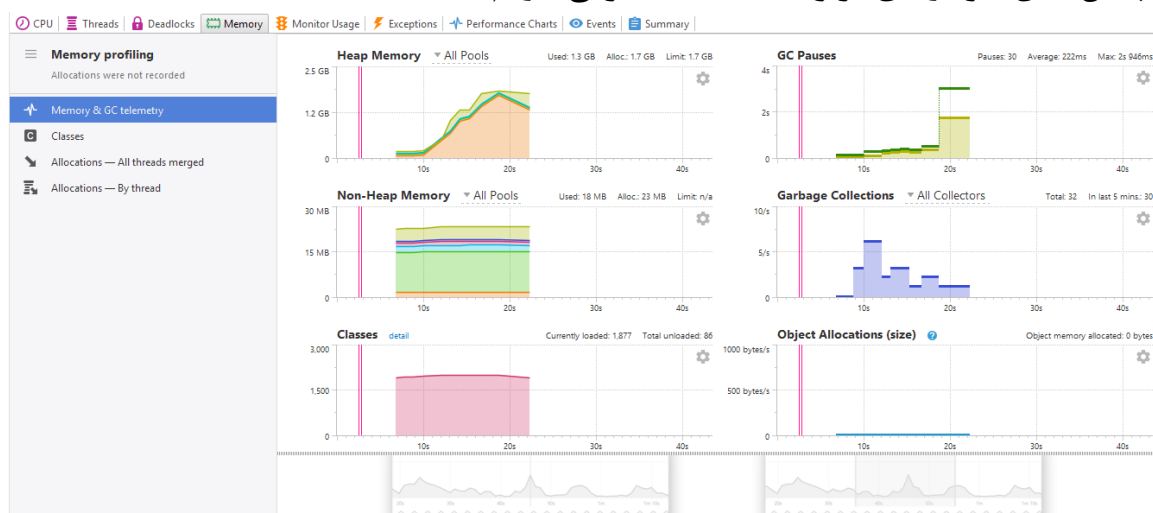
Call Tree	Time (ms)	Samples
main Group: 'main' Native ID: 9476	4,203	375
JavaCup.java:14 JavaCup.temp()	4,171	342
JavaCup.java:29	3,921	168
JavaCup.java:30 java.util.ArrayList.add(Object)	2,718	147
JavaCup.java:30 java.lang.Integer.valueOf(int)	1,031	10
Integer.java:1050	156	10
JavaCup.java:30	15	1
JavaCup.java:7 java.util.Scanner.<init>(InputStream)	156	8
JavaCup.java:7 java.util.Scanner.<clinit>()	78	4
JavaCup.java:9 java.util.Scanner.nextInt()	15	66
JavaCup.java:11 java.util.Scanner.nextInt()	0	59
JavaCup.java:13 java.util.Scanner.nextInt()	0	37
sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, String)	31	33
Monitor Ctrl-Break Group: 'main' Native ID: 5992	125	375
com.intellij.execution.application.AppMainV2\$1.run()	125	375

شکل ۶ - نمودار درختی thread ها و زمان اجرای توابع (قبل از بهبود)

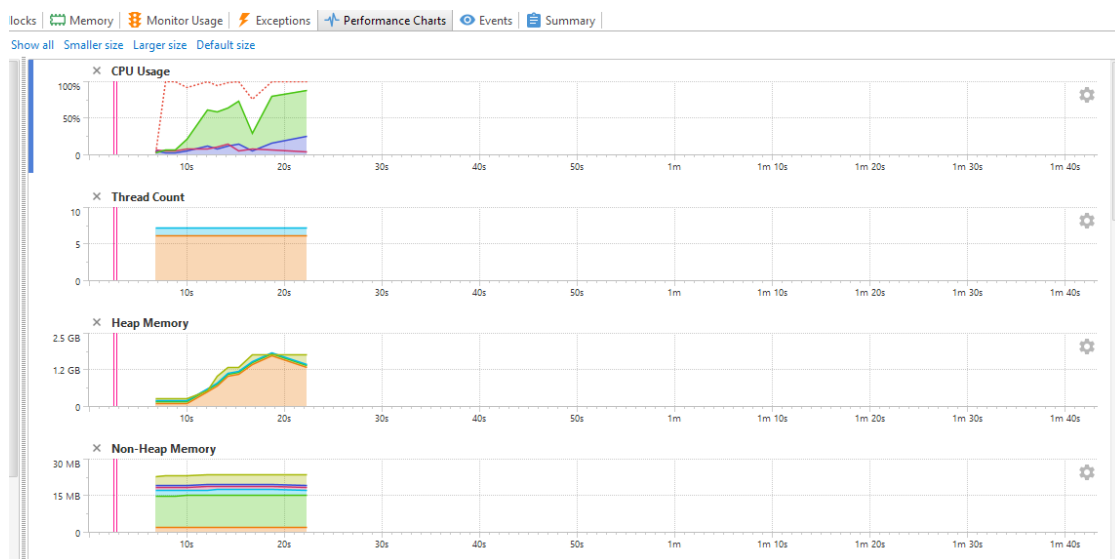
Method	Time (ms)	Own Time (ms)	Samples
JavaCup.main(String[]) JavaCup.java	4,171 96 %	0	342
JavaCup.temp() JavaCup.java	3,921 91 %	2,734	168
java.util.ArrayList.add(Object) ArrayList.java	1,031 24 %	1,031	10
java.lang.Integer.valueOf(int) Integer.java	156 4 %	156	10
java.util.Scanner.<init>(InputStream) Scanner.java	156 4 %	156	8
com.intellij.rt.execution.application.AppMainV2\$1.run() AppMainV2.java	125 3 %	125	375
java.util.Scanner.<clinit>() Scanner.java	78 2 %	78	4
sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, String) LauncherHelper.java	31 1 %	31	33
java.util.Scanner.nextInt() Scanner.java	15 0 %	15	162

شکل ۷- نمودار زمان مصرفی و درصد زمانی مربوط به توابع (قبل از بهبود)

همچنین طبق نمودارهای مربوط به حافظه مصرفی داریم:

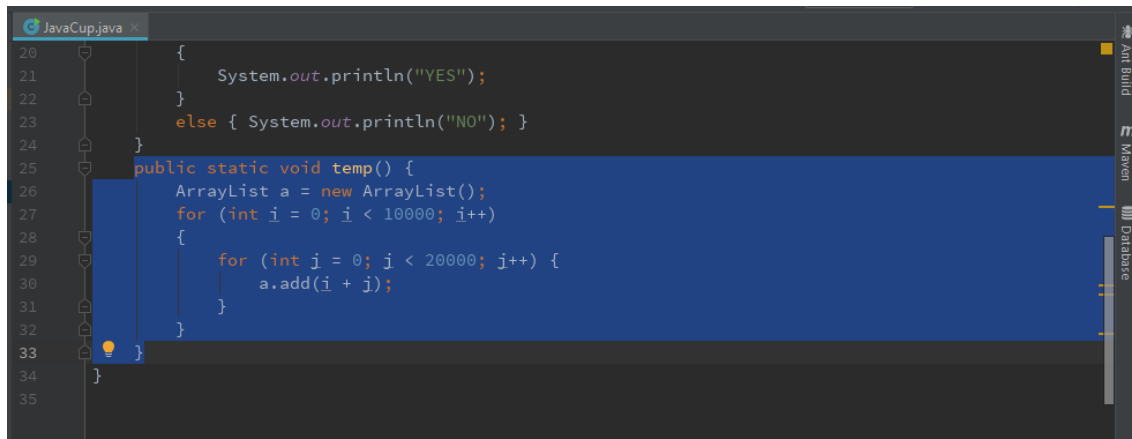


شکل ۸- نمودار حافظه مصرفی (قبل از بهبود)



شکل ۹- نمودار کارایی پردازنده و حافظه و threadها (قبل از بهبود)

طبق بررسی نمودارهای بالا و حجم بالای مصرف منابع و زمان اجرایی توسط تابع `temp`، به بررسی دقیقتر آن میپردازیم.



شکل ۱۰- تابع `temp` (قبل از بهبود)

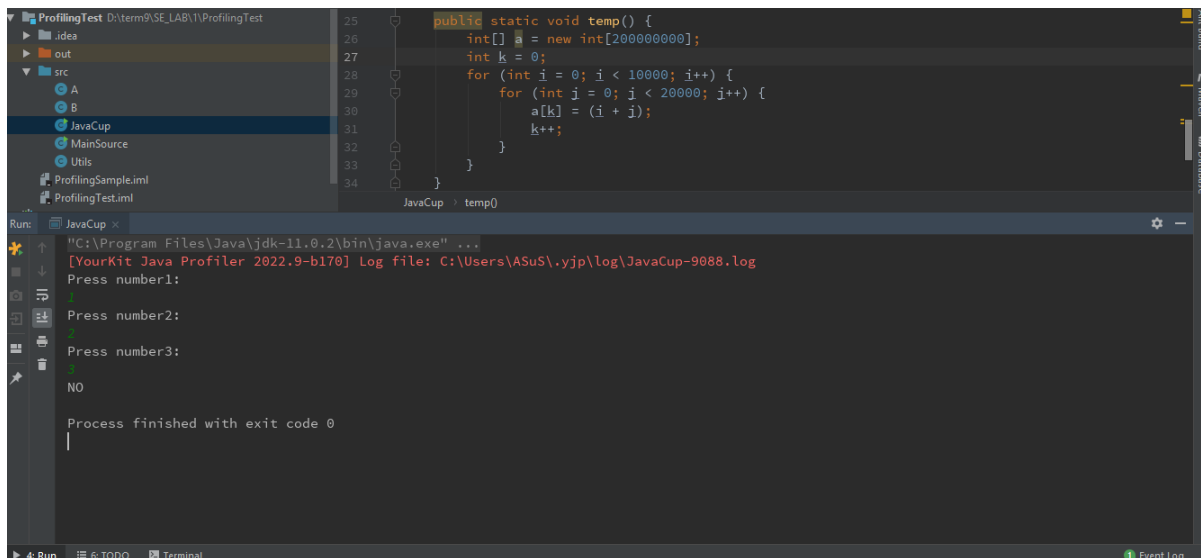
در این تابع، از `ArrayList` جاوا استفاده شده و $10000 * 20000$ مرتبه به آن عضوی اضافه می‌کند. این داده ساختار به صورت فضای پویا پیاده‌سازی شده و هر بار که فضای آن تکمیل شود، سائز خود را دو برابر می‌کند. به همین علت منابع و زمان اجرایی بالایی را مصرف کرده و `heap` پر می‌شود (خطای پس از اجرای برنامه نیز به همین علت بود).

برای بهبود و جلوگیری از این مشکل، میتوان از داده ساختار لیست با طول مشخص استفاده کنیم و مجدداً عملیات `profiling` را اجرا می‌کنیم.

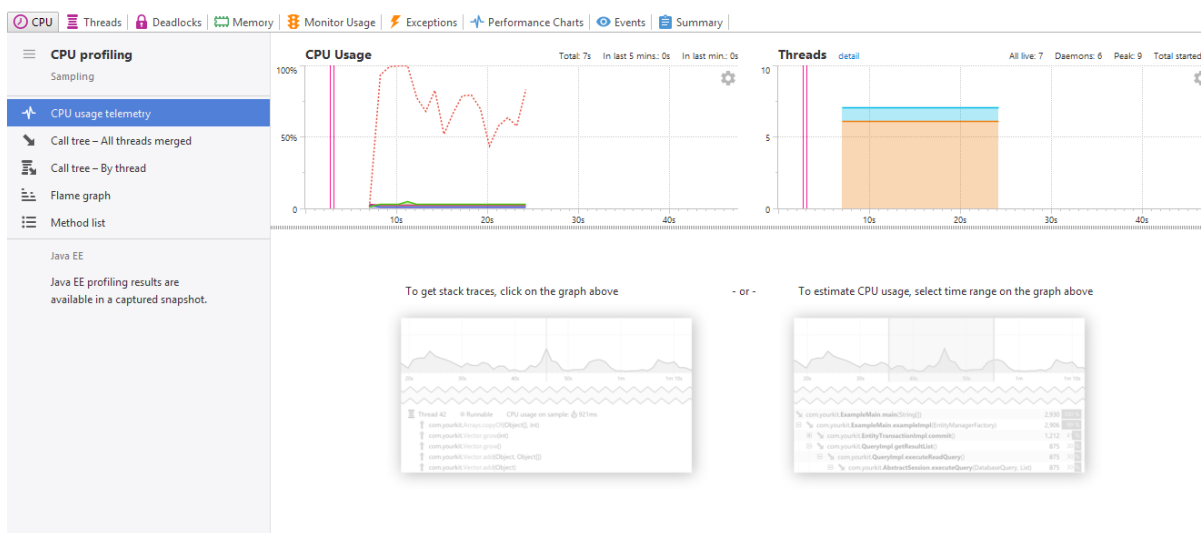


شکل ۱۱- تابع `temp` (پس از بهبود)

پس از اعمال این تغییر برنامه بدون خطا اجرا شد و داریم:

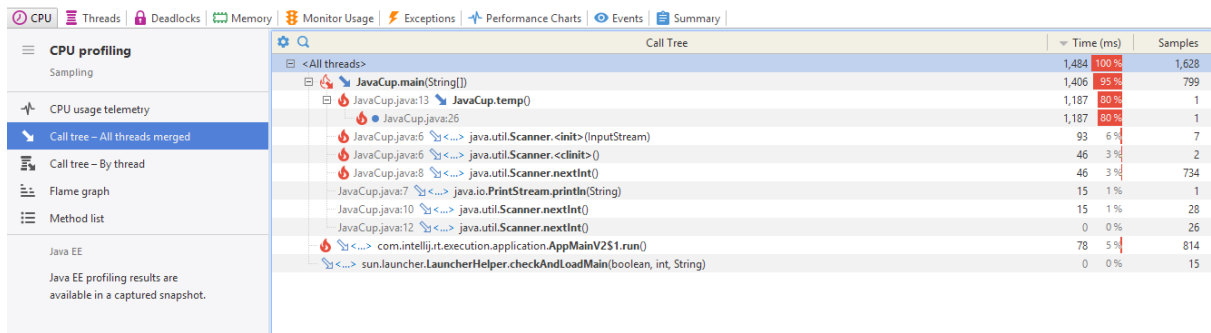


شکل ۱۲ - خروجی برنامه بعد از اجرای عملیات *profiling* (پس از بهبود)

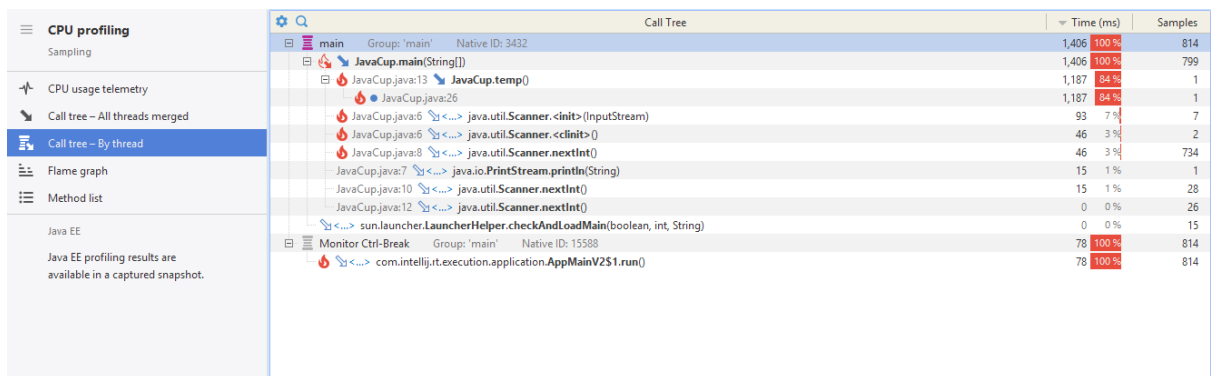


شکل ۱۳ - نمودار استفاده از پردازنده (پس از بهبود)

مشاهده می شود زمان اجرایی مصرف شده توسط تابع **temp** به 80 درصد کاهش یافت.



شکل ۱۴ - نمودار درختی مربوط به *thread* ها (پس از بهبود)

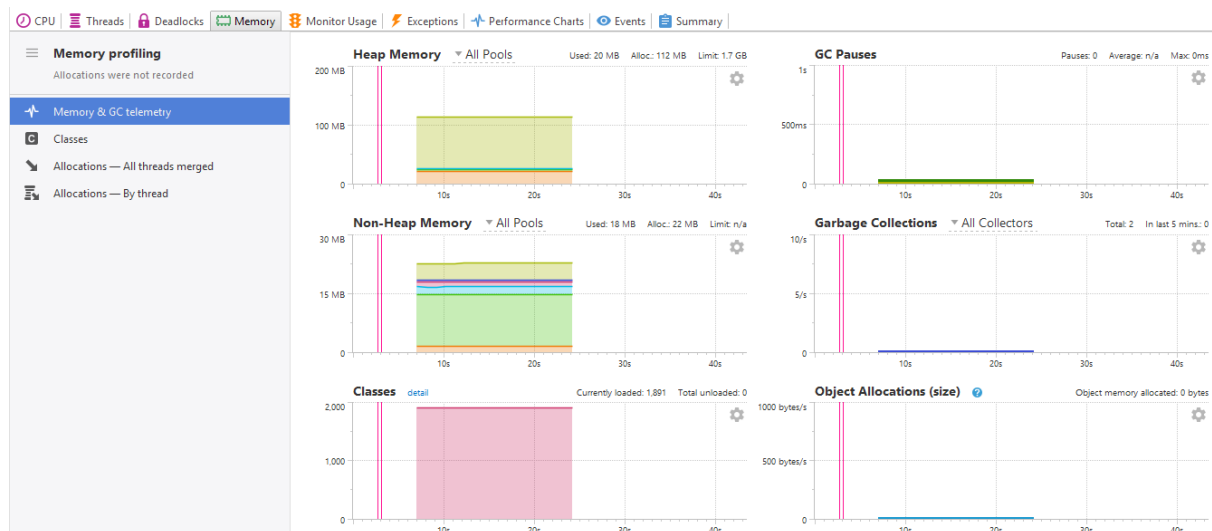


شکل ۱۵ - نمودار درختی *thread* ها و زمان اجرای توابع (پس از بهبود)

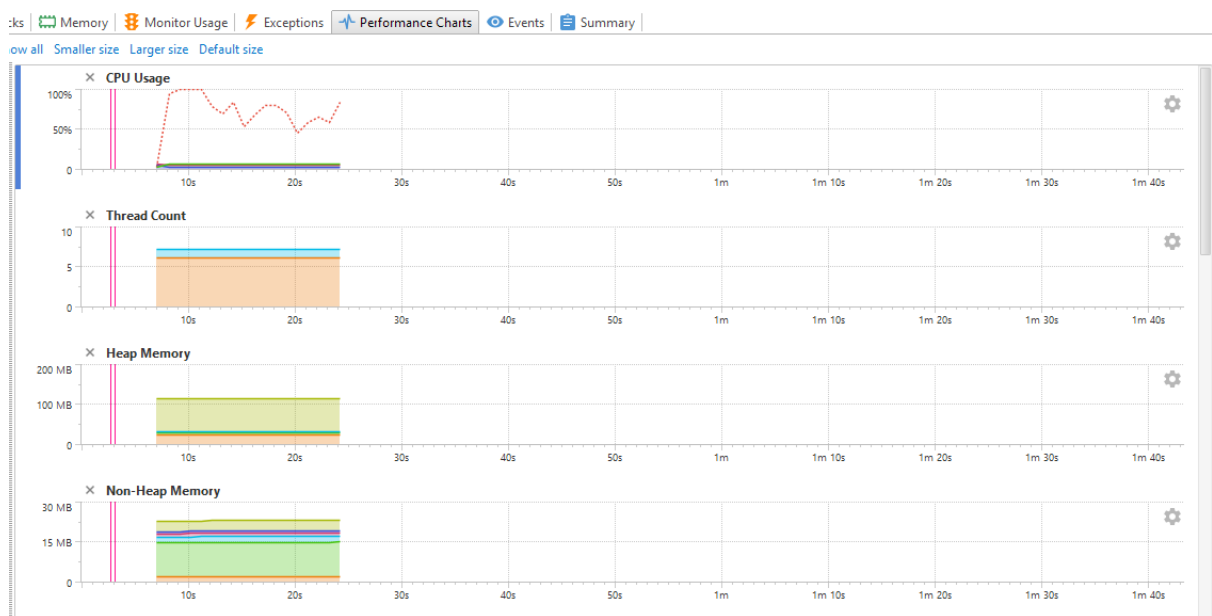
Method	Time (ms)	Own Time (ms)	Samples
JavaCup.main(String[]) JavaCup.java	1,406 95 %	0	799
JavaCup.temp() JavaCup.java	1,187 80 %	1,187	1
java.util.Scanner.<init>(InputStream) Scanner.java	93 6 %	93	7
com.intellij.rt.execution.application.AppMainV2\$1.run() AppMainV2.java	78 5 %	78	814
java.util.Scanner.nextInt() Scanner.java	62 4 %	62	788
java.util.Scanner.<clinit>() Scanner.java	46 3 %	46	2
java.io.PrintStream.println(String) PrintStream.java	15 1 %	15	1
sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, String) LauncherHelper.java	0 0 %	0	15

شکل ۱۶ - نمودار زمان مصرفی و درصد زمانی مربوط به توابع (پس از بهبود)

همچنین طبق نمودارهای مربوط به حافظه مصرفی داریم:



شکل ۱۷ - نمودار حافظه مصرفی (پس از بهبود)



شکل ۱۸ - نمودار کارایی پردازنده و حافظه و threadها (پس از بهبود)

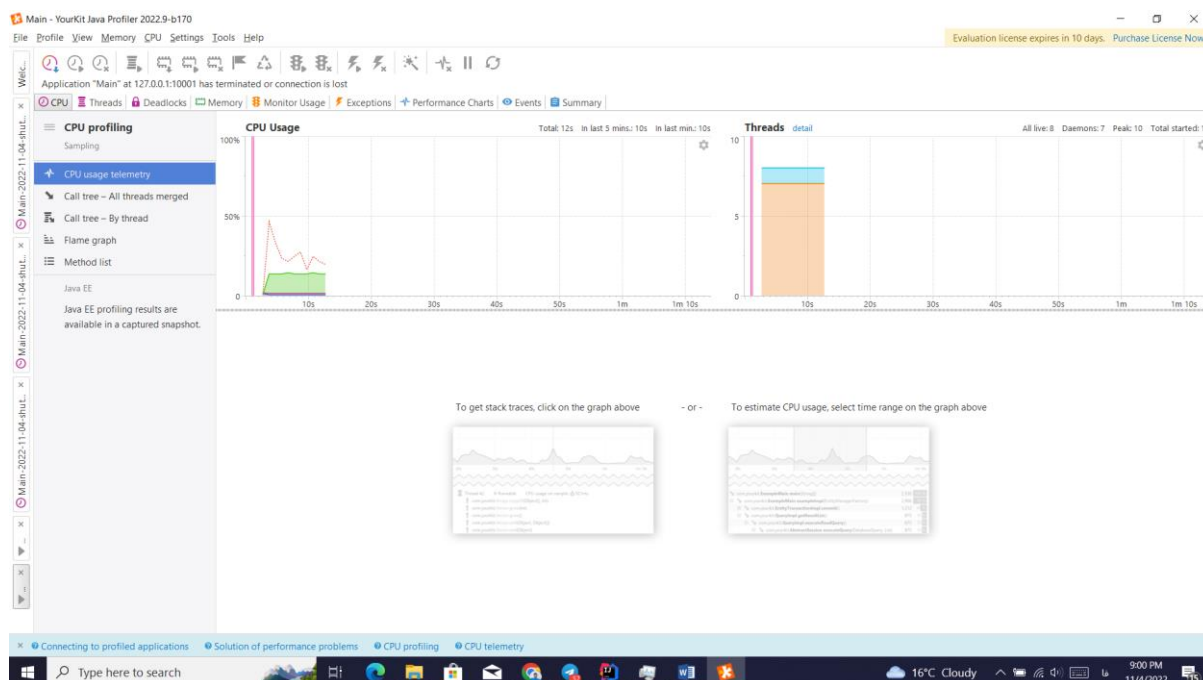
طبق نمودارهای بالا، مشاهده می‌شود پس از تغییر تابع **temp** و استفاده از داده ساختار مناسب، منابع و زمان کمتری مصرف شد و میزان **heap** استفاده شده از 1.3GB به 20MB رسید که بهبود قابل توجه‌ای محسوب می‌شود. همچنین درصد زمان اجرایی تابع **temp** نیز کاهش یافت.

بخش دوم.

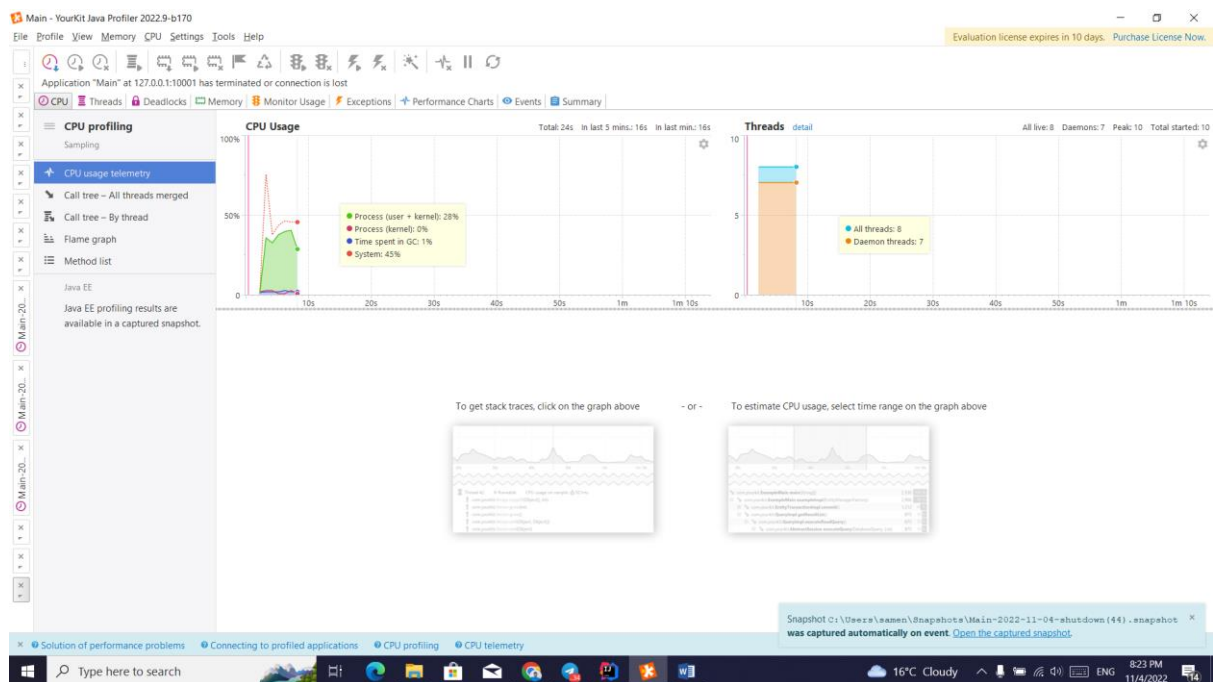
در این بخش 2 کد مختلف برای الگوریتم مرتب سازی داریم. کد اول مربوط به الگوریتم مرتب سازی حبابی (Bubble Sort) می باشد که مرتبه $O(n^2)$ دارد و کد دوم مربوط به الگوریتم مرتب سازی ادغامی (Merge Sort) می باشد که مرتبه $O(n \log n)$ دارد. به دلیل این که میخواستیم کد مرج سورت حداقل 10 ثانیه اجرا شود طول آرایه را برابر با عدد 9000000 قرار دادیم و این عدد برای اجرا در حالت بابل سورت نیاز به ساعت ها اجرا دارد و به همین دلیل در کد بابل سورت این عدد را 20000 قرار دادیم.

حال نوبت به نتایج اجرا می رسد (عکس اول مربوط به بابل سورت و عکس دوم مربوط به مرج سورت است).

CPU usage telemetry



شکل ۱۹ - نمودار CPU usage telemetry در الگوریتم مرتب سازی حبابی



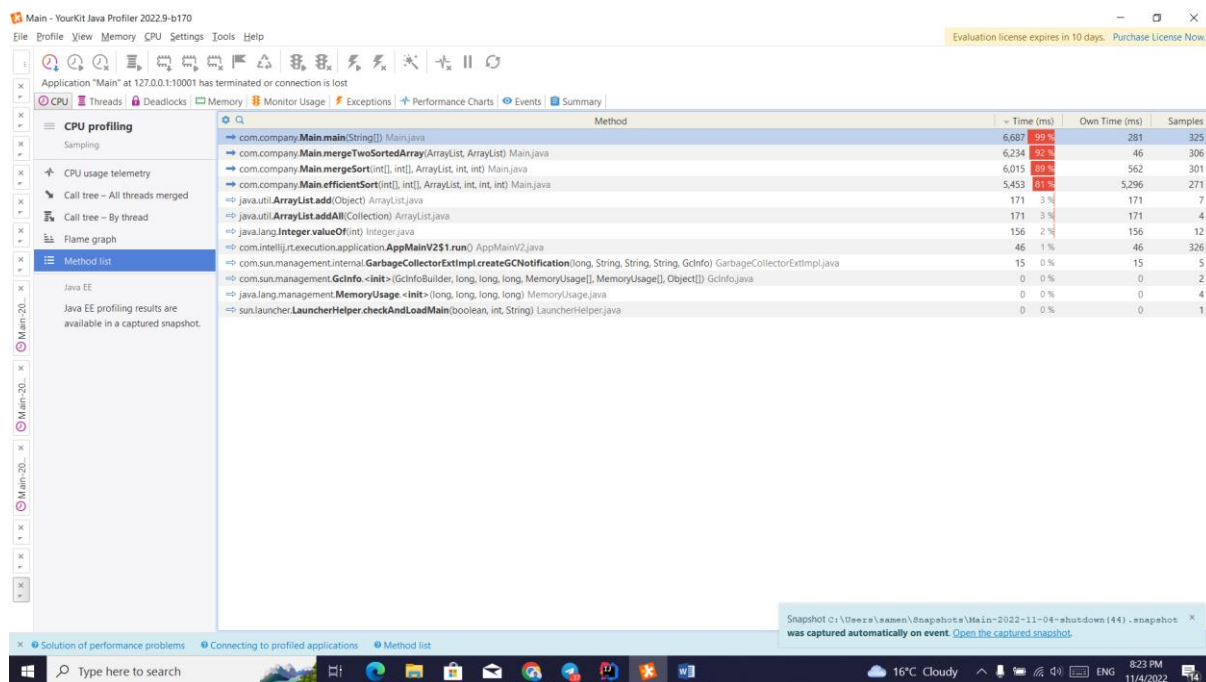
شکل ۲۰- نمودار *CPU usage telemetry* در الگوریتم مرتب‌سازی ادغامی

همانطور که معلوم است الگوریتم مرتب‌سازی ادغامی به طور بهینه‌تری از قدرت CPU استفاده می‌کند که نشان از بهتر بودنش است. تا حدی که دوست داریم کارها را موازی جلو ببریم.

:Method List

Method	Time (ms)	Own Time (ms)	Samples
com.company.Main.bubbleSort(ArrayList) Main.java	10,515	10,234	503
com.company.Main.main(String[]) Main.java	10,515	0	504
com.company.Main.mergeTwoSortedArray(ArrayList, ArrayList) Main.java	10,515	0	503
java.lang.Integer.valueOf(int) Integer.java	234	234	16
java.io.PrintStream.println(int) PrintStream.java	46	46	2
com.intellij.execution.application.AppMainV2\$1.run() AppMainV2.java	31	31	504
com.sun.management.GcInfo.<init>() GcInfo.java	0	0	2
com.sun.management.internal.GarbageCollectorExtImpl.createGCNotification(long, String, String, GcInfo) GarbageCollectorExtImpl.java	0	0	1
java.lang.management.MemoryUsage.<init>() MemoryUsage.java	0	0	1

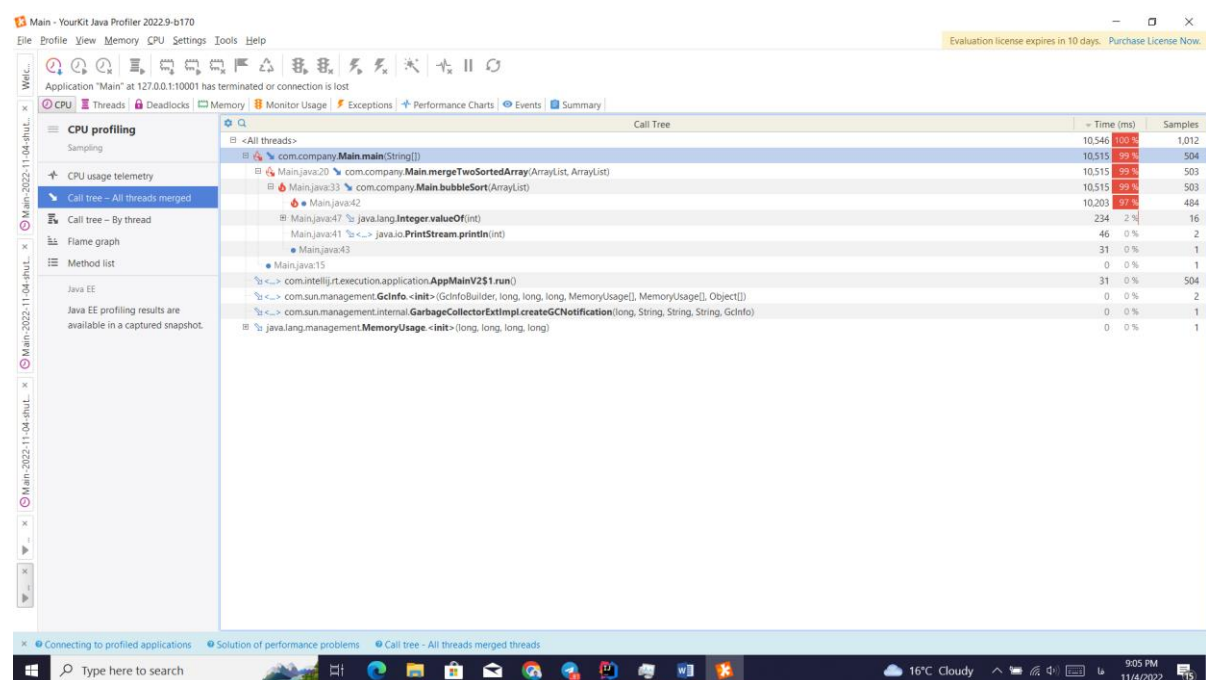
شکل ۲۱- نمودار زمان مصرفی و درصد زمانی مربوط به مرتب‌سازی حبابی



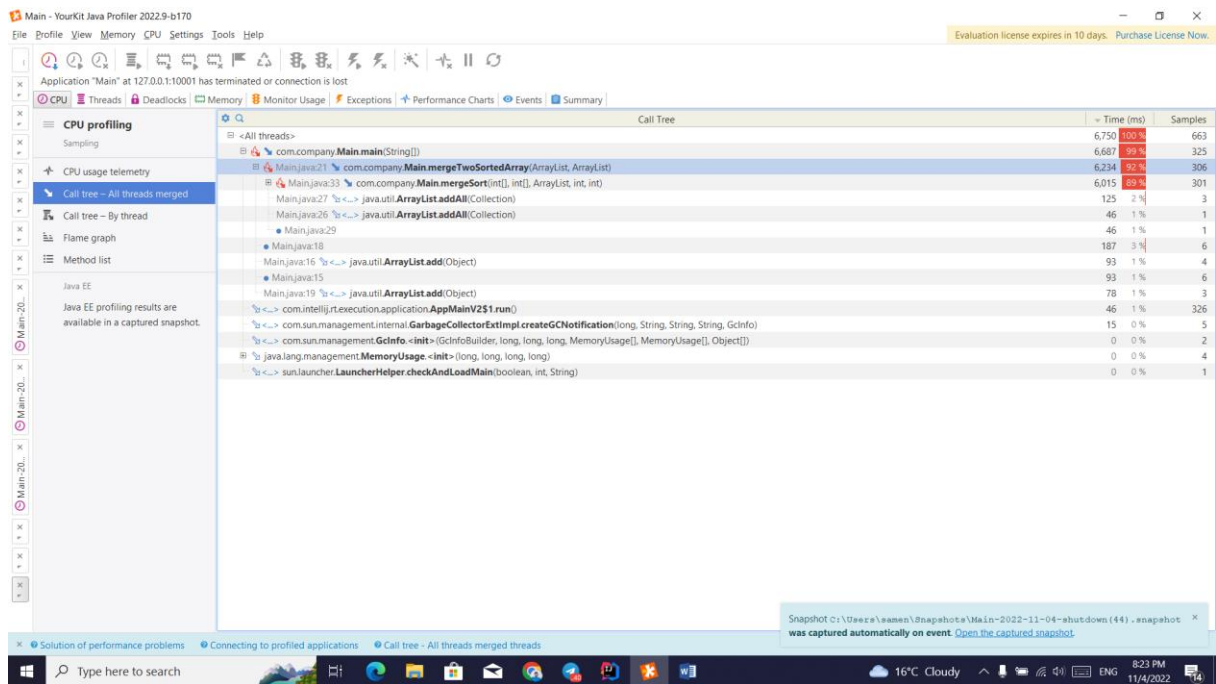
شکل ۲۲- نمودار زمان مصرفی و درصد زمانی مربوط به مرتب‌سازی ادغامی

طبق نمودارهای بالا، درصد زمان مصرفی استفاده شده توسط تابع sort در حالت دوم به مراتب بهتر از حالت اول عمل کرده است که به علت پیچیدگی زمانی کمتر آن است.

:Call Tree

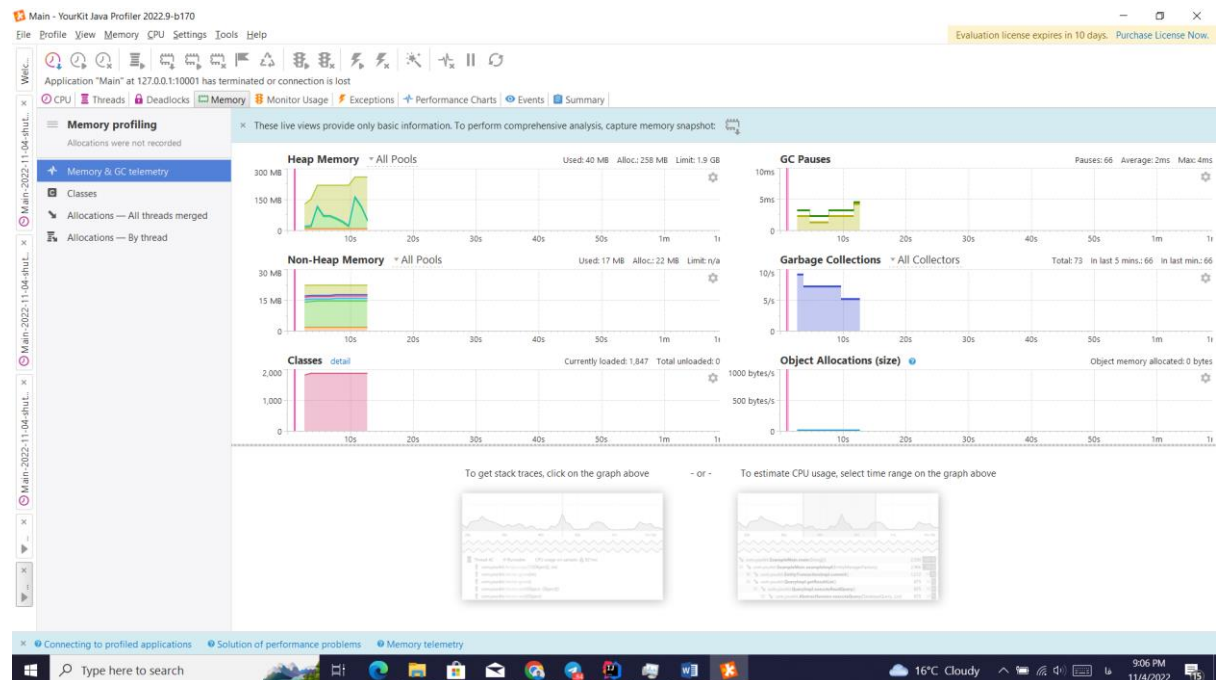


شکل ۲۳- نمودار درختی thread ها و زمان اجرای توابع در الگوریتم مرتب‌سازی حبابی

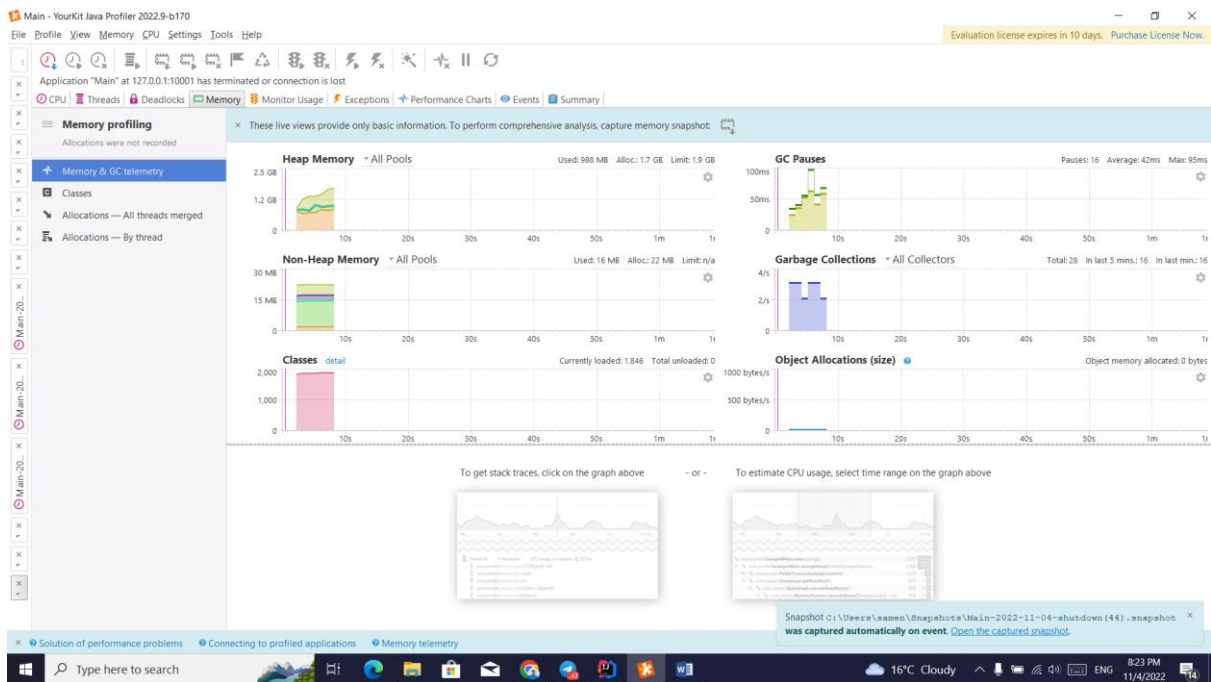


شکل ۲۴ - نمودار درختی thread ها و زمان اجرای توابع در الگوریتم مرتب‌سازی ادغامی

Memory



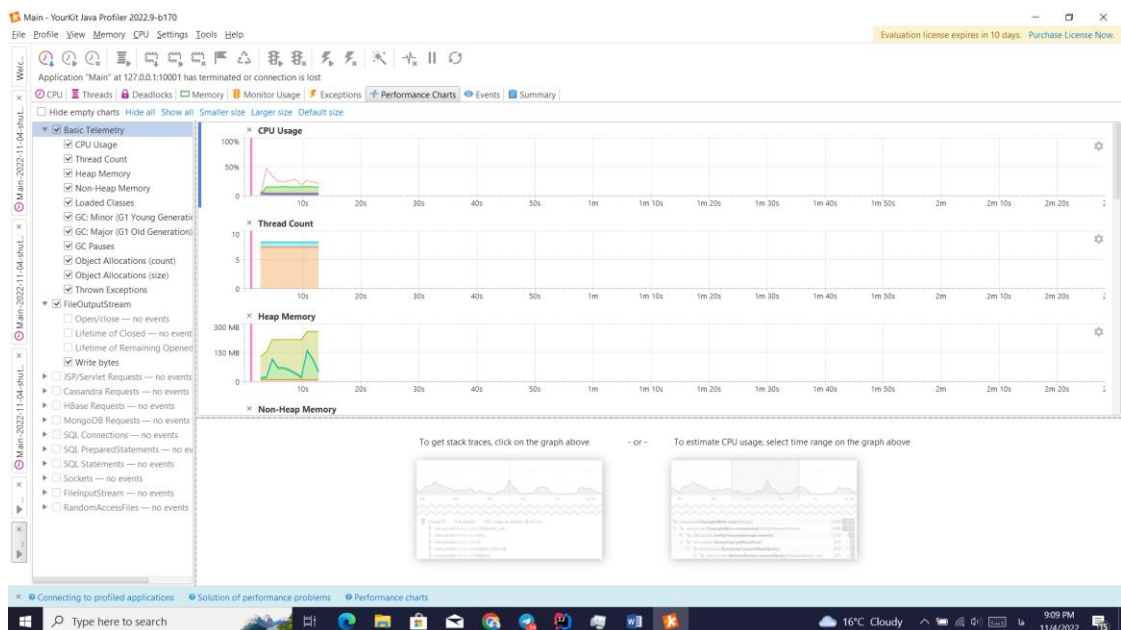
شکل ۲۵ - نمودار حافظه مصرفی در حالت مرتب‌سازی حبابی



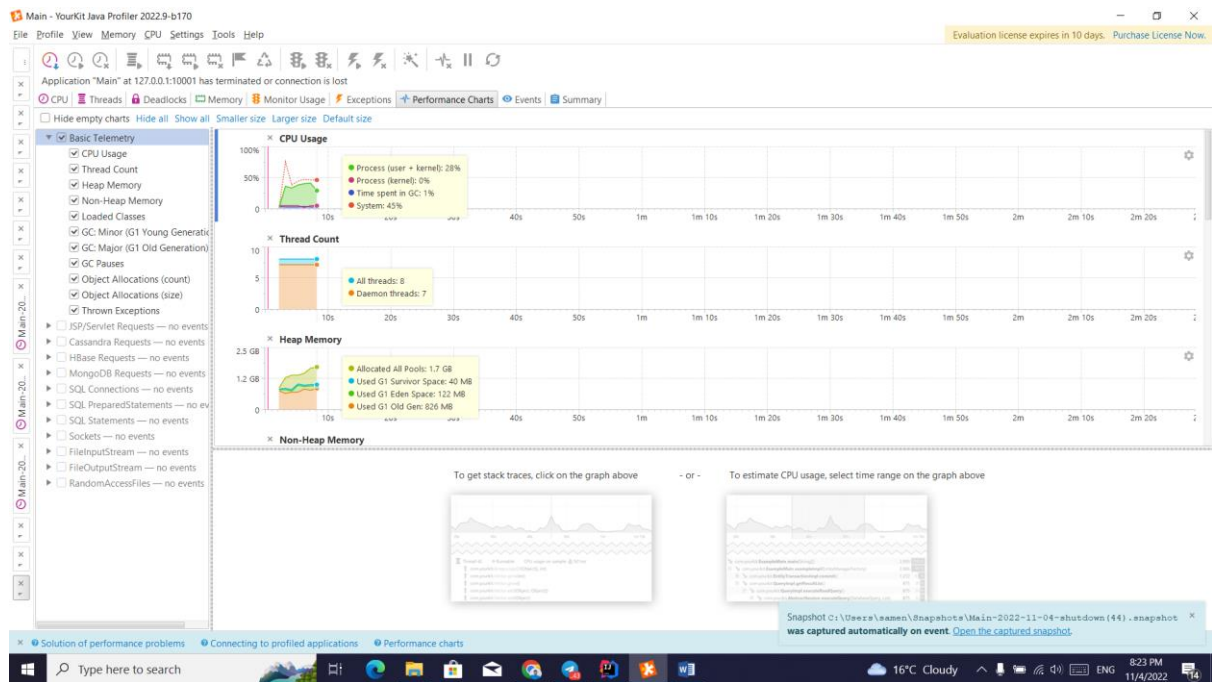
شکل ۲۶- نمودار حافظه مصرفی در حالت مرتب‌سازی ادغامی

روش مرج سورت از حافظه ی هیپ بیشتری استفاده می کند که دلیلش صدا کردن تابع های بیشتر است. همچنین GC Pauses بیشتری دارد تقریبا 10 برابر. اما نمودار مربوط به Garbage collection اش کمتر است.

:Performance Charts



شکل ۲۷ - نمودار کارایی پردازنده و حافظه و threadها در حالت مرتب‌سازی حبابی



شکل ۲۸ - نمودار کارایی پردازنده و حافظه و threadها در حالت مرتب‌سازی ادغامی

در کل دیده می شود که CPU Usage و Heap Memory در کد دوم که از الگوریتم مرتب‌سازی ادغامی استفاده می کند بهینه تر است.