Aula 7
Análise de Complexidade de Algoritmos

Algoritmos e Estruturas de Dados

Análise

de Complexidade de Algoritmos

Análise de Algoritmos

- Algoritmo
  - ***Def:*** conjunto ou sequência finita de instruções/ações a seguir para resolver um problema

- Análise de algoritmos
  - Provar que um algoritmo está correcto
  - Determinar recursos exigidos por um algoritmo

    Tempo, espaço

    Prever o crescimento dos recursos exigidos por um algoritmo à medida que o tamanho dos dados de entrada cresçe.

Complexidade de algoritmos

- Complexidade temporal
  - ***Def:*** tempo que um programa ou algoritmo demora a executar, em função do tamanho ou complexidade (n) da entrada

    T(n) – tempo de execução em função de n

- Complexidade espacial
  - ***Def:*** espaço de memória que um programa ou algoritmo necessita para executar até ao fim, em função do tamanho ou complexidade (n) da entrada.

    S(n) – espaço de memória exigido em função de n

Análise de Complexidade

- 3 formas de análise de complexidade
  - Análise empírica
  - Análise por modelos matemáticos
  - Análise assimptótica

Análise

empírica

Exemplo: 3-Sum

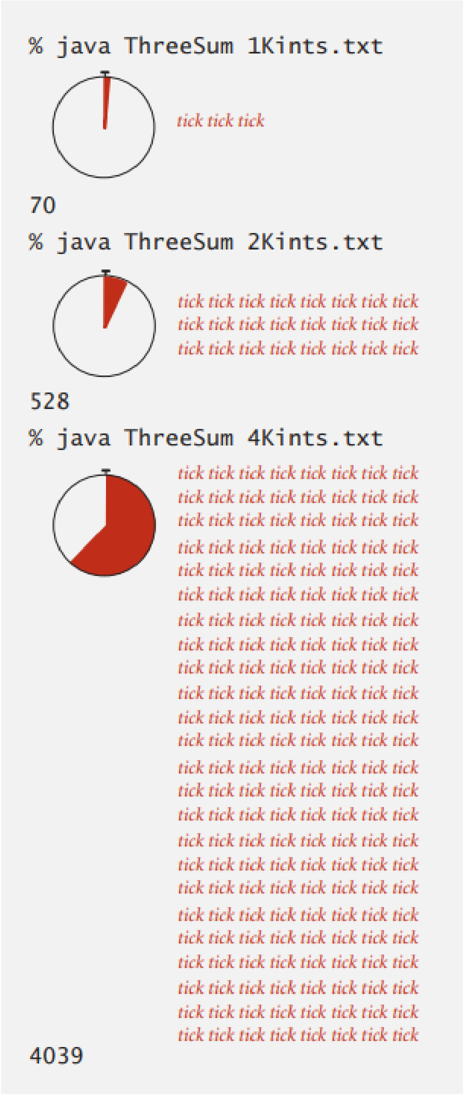
- Objectivo
  - Dado um número n de inteiros (recebidos num array)
  - Determinar quantas combinações de 3 elementos somam 0

```
public static int threeSum(int[] a)
{
   int n = a.length;
   int count = 0;
   for(int i = 0; i < n; i++)
   {
      for(int j = i+1; j < n; j++)
      {
         for(int k = j+1; k < n; k++)
         {
               if(a[i]+a[j]+a[k] == 0)
                  count++;
         }
      }
   }
   return count;
}
```

Testes empíricos

- Medir o tempo de execução médio de um algoritmo
  - Podemos faze-lo de forma manual
  - Ou usando ferramentas de *profiling*

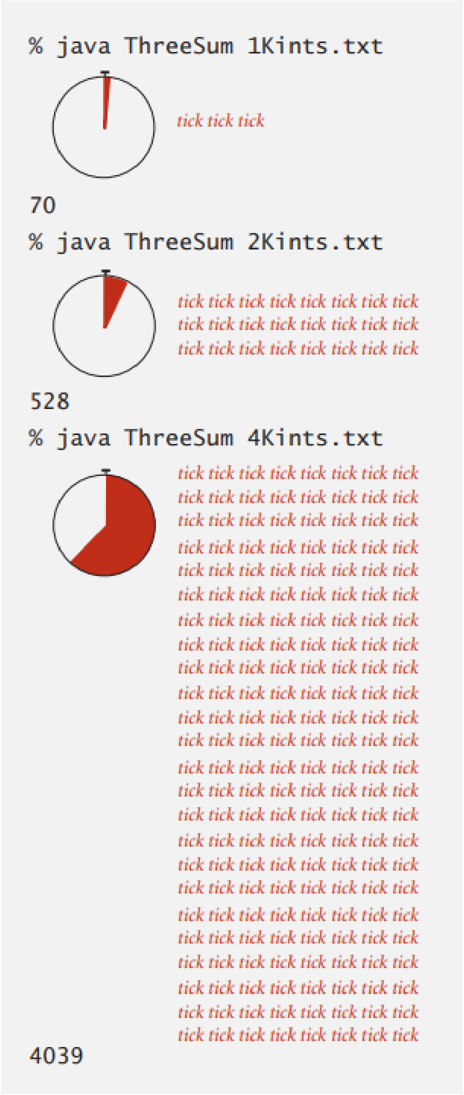
    Ex: Java Microbenchmark Harness (JMH)



Contabilização do tempo

- Ex: Classe StopWatch

```
StopWatch watch = new StopWatch();
watch.start();
…
watch.stop();
System.out.println("Time Elapsed: " +
watch.getTime());
```



Contabilização do tempo

- Ex: Teste Mooshak que testa a eficiência do método m de um objecto o

```
private static long getAverageCPUTimeOnObjectMethod(Object o)
{
    long startTime,stopTime;
    long elapsedCPU = 0;
    int tests = 50;
    ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
    long[] allThreadIds = threadMXBean.getAllThreadIds();

    for(int i = 0; i < tests; i++)
    {
        startTime = getCPUTime(threadMXBean,allThreadIds);
        o.m();
        stopTime = getCPUTime(threadMXBean,allThreadIds);
        elapsedCPU += stopTime - startTime;
    }

    elapsedCPU /= tests;

    return elapsedCPU;
}
```

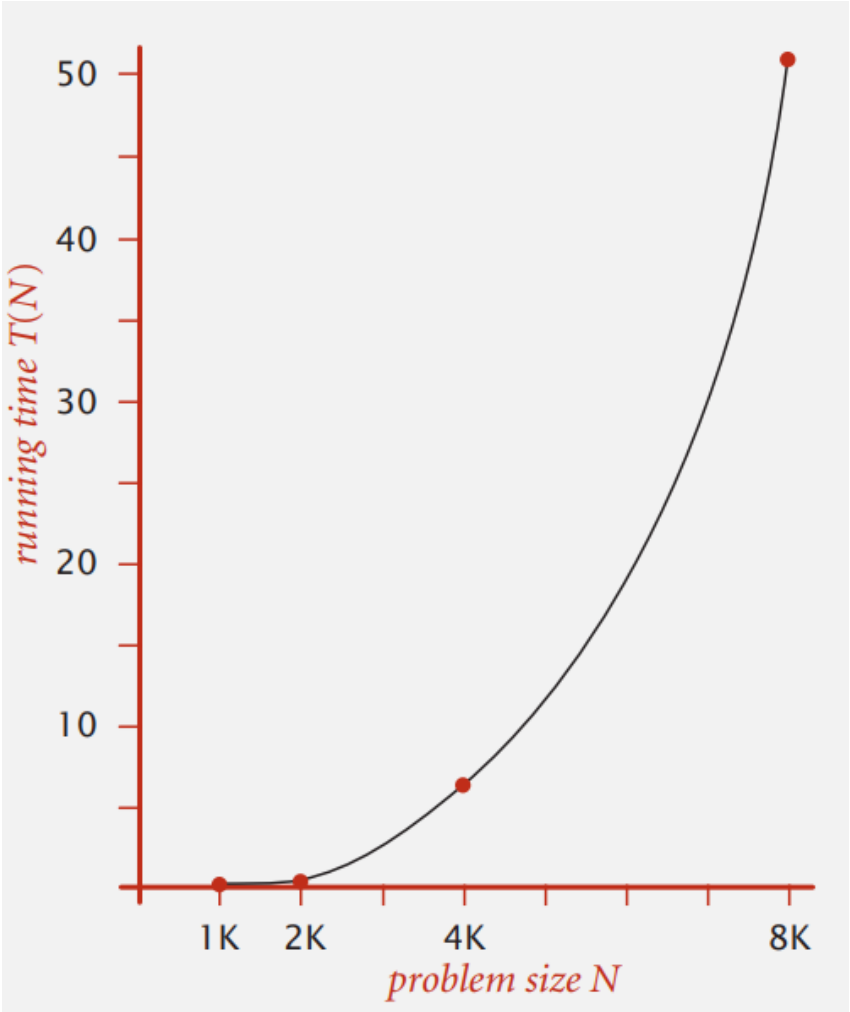
Contabilização do tempo

- Correr um algoritmo para vários *inputs* e registrar o tempo

n	Tempo (segundos)
250	0.0
500	0.0
1 000	0.1
2 000	0.8
4 000	6.4
8 000	51.1
16 000	?

Análise empírica

- Utilizar os dados registados para desenhar um gráfico
- Estimar a curva



Ensaios de razão dobrada

- Técnica para determinar de forma empírica a ordem de crescimento de um algoritmo

Ensaio de razão dobrada

1) Testar um algoritmo à medida que se dobra a complexidade de n

- n=250
- n=500
- n=1000
- n=2000
- n=4000
- …

Ensaio de razão dobrada

- 2) calcular a razão dobrada r
- Razão dobrada $\mathbf{r} = \frac{T(2n)}{T(n)}$
- Razão entre o tempo de execução para *2n* e o tempo de execução para *n*

Exemplo: 2-sum

```
public static int twoSum(int[] a)
{
    int n = a.length;

    int count = 0;
    for(int i = 0; i < n; i++)
    {
        for(int j = i+1; j < n; j++)
        {
            if(a[i] + a[j] == 0)
                count++;
        }
    }

    return count;
}
```

Exemplo: 2-sum

```
public static double calculateAverageExecutionTime(int n)
{
    int trials = 30;
    double totalTime = 0;

    for(int i = 0; i < trials; i++)
    {
        int[] example = generateExample(n);
        long time = System.currentTimeMillis();
        twoSum(example);
        totalTime += System.currentTimeMillis() - time;
    }

    return totalTime/trials;
}
```

```
public static int[] generateExample(int n)
{
    Random r = new Random();
    int [] examples = new int[n];

    for(int i = 0;  i < n; i++)
    {
        examples[i] = r.nextInt();
    }

    return examples;
}
```

Exemplo: 2-sum

```
public static void main(String[] args)
{
    int n = 125;
    double previousTime = calculateAverageExecutionTime(n);
    double newTime;
    double doublingRatio;
    for(int i = 250; true; i*=2)
    {
        newTime = calculateAverageExecutionTime(i);
        if(previousTime > 0)
        {
            doublingRatio = newTime/previousTime;
        }
        else doublingRatio = 0;

        previousTime = newTime;
        System.out.println(i + "\t" + newTime + "\t" + doublingRatio);
    }
}
```

Ensaio de razão dobrada

3) Esperar que o valor de razão dobrada estabilize para n grande

4) Transformar o valor de r numa potência *b* de 2

- $r = 2^b (=) \log_2 r = \log_2 2^b (=) b = \log_2 r$

5) A complexidade temporal *T(n)* é aproximada por

$T(n) \sim \boldsymbol{n^b}$

Ensaio de razão dobrada

4) Transformar o valor de r numa potência *b* de 2

- $r = 2^b (=) \log_2 r = \log_2 2^b (=) b = \log_2 r$

5) A complexidade temporal *T(n)* é aproximada por

$T(n) \sim n^b$

Exemplo 2-sum:

$r \sim 4 = 2^2$

$T(n) \sim n^2$

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe"
250 0.13333333333333333 1.3333333333333333
500 0.06666666666666667 0.5
1000    0.13333333333333333 2.0
2000    0.4 3.0
4000    1.8666666666666667  4.666666666666666
8000    7.733333333333333   4.142857142857143
16000   30.233333333333334  3.90948275862069
32000   123.63333333333334  4.089305402425579
```

n | *tempo execução* | *r*

Ensaio de razão dobrada

4) Transformar o valor de r numa potência *b* de 2

- $r = 2^b (=) \log_2 r = \log_2 2^b (=) b = \log_2 r$

5) A complexidade temporal *T(n)* é aproximada por

$T(n) \sim \boldsymbol{n^b}$

Exemplo 3-sum:

$r \sim 8 = 2^{3}$

$T(n) \sim n^3$

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe"
250 0.7 1.7499999999999998
500 5.733333333333333   8.190476190476192
1000    45.46666666666667   7.930232558139535
2000    339.1666666666667   7.459677419354839
```

n | *tempo execução* | *r*