O problema do 3*n + 1 - Otimização

Este documento apresenta uma breve descrição e análise comparativa de 12 implementações do problema 3*n+1.

Id	Nome do Arquivo	Descrição Geral	Características de Otimização
Simples	**P3nMais1_Simples.java**	Algoritmo simples (seguindo o enunciado).	nenhuma
Rec	**P3nMais1_Recursivo.java**	Algoritmo recursivo para resolver o problema 3n+1	nenhuma
Hash	**P3nMais1_Hash.java**	Algoritmo simples com a adição de um *hash* para evitar o recálculo de valores.	1. hash
Hash2	**P3nMais1_Hash2.java**	Uso de hash guardando não só as soluções pré-calculadas mas também as soluções parciais	1. hash 2. armazenamento das soluções parciais
Hash2s	**P3nMais1_Hash2_semArray.java**	Uso de hash guardando não só as soluções pré-calculadas mas também as soluções parciais e uso de um arranjo ao invés de um ArrayList.	1. hash 2. armazenamento das soluções parciais 3. uso de arranjo ao invés de ArrayList
RecHash	**P3nMais1_RecursivoHash.java**	Uso do hash no algoritmo recursivo (não é necessário o uso de arranjo ou ArrayList).	1. hash 2. armazenamento das soluções parciais
Duplo	**P3nMais1_PassoDuplo.java**	Algoritmo simples, porém, sempre que um número é ímpar executado um "passo duplo": ao invés: n = 3*n+1 e temp++; utiliza: n=(3*n+1)/2 e temp+=2;	1. "passo duplo"
RecPD	**P3nMais1_RecursivoPassoDuplo.java**	Algoritmo recursivo utilizando o passo duplo.	1. "passo duplo"
HashPD	**P3nMais1_HashPassoDuplo.java**	Uso de hash no algoritmo que utiliza o passo duplo.	1. hash 2. "passo duplo"
Hash2PD	**P3nMais1_Hash2PassoDuplo.java**	Uso de hash guardando não só as soluções pré-calculadas mas também as soluções parciais e utiliza o passo duplo.	1. hash 2. armazenamento das soluções parciais 3. "passo duplo"
Hash2sD	**P3nMais1_Hash2_semArrayPassoDuplo.java**	Uso de hash guardando não só as soluções pré-calculadas mas também as soluções parciais; uso de um arranjo ao invés de um ArrayList e uso do passo duplo.	1. hash 2. armazenamento das soluções parciais 3. uso de arranjo ao invés de ArrayList 4. "passo duplo"
RecHPD	**P3nMais1_RecursivoHashPassoDuplo.java**	Uso de hash em um algoritmo recursivo guardando não só as soluções pré-calculadas mas também as soluções parciais e utiliza o passo duplo.	1. hash 2. armazenamento das soluções parciais 3. "passo duplo"

Características de Otimização e Implementação

Hash:

Abaixo foram destacadas as linhas de código envolvidas com o uso do hash (utilizando a classe Hashtable):

```
public static int executar(int ini, int fim){
      Hashtable<Integer,Integer> log = new Hashtable<Integer,Integer>();
      int n;
      int temp;
      int maximo;
      maximo = 0;
      for (int i=ini;i<=fim;i++){
            n = i;
            temp = 1;
            while (n > 1){
                  if (log.containsKey(n)){
                        temp+=log.get(n)-1;
                        n=1;
                  }else{
                        temp++;
                        if (n % 2 == 0) n = n/2;
                        else n = 3*n+1;
                  }
            }
            log.put(i, new Integer(temp));
            if (temp>maximo) maximo = temp;
      }
      return maximo;
}
```

Passo Duplo:

```
            while (n > 1){
                  if (n % 2 == 0) {
                        n = n/2;
                        temp++;
                  }
                  else{
                        n = (3*n+1)/2;
                        temp+=2;
                  }
            }
```

Recursivo:

```
public static int executar(int ini, int fim){
	int temp;
	int maximo;
	maximo = 0;
	for (int i=fim;i>=ini;i--){
		temp = recursivo(i);
		if (temp>maximo) maximo = temp;
	}
	return maximo;
}
```

```
static int recursivo(int n){
	if (n == 1) return 1;
	int temp = 1;
	if (n % 2 == 0) {
		temp += recursivo(n/2);
	}else{
		temp += recursivo(3*n+1);
	}
	return temp;
}
```

A Figura 3 apresenta o número de soluções armazenadas nos *hashs* para cada uma das soluções (considerando a execução utilizando o intervalo de 1 a 100.000). Note que esse número é proporcional a memória gasta em cada implementação. Ao observar as figuras anteriores é possível notar que as implementações que guardam mais soluções (os resultados parciais) acabam por apresentar um desempenho de tempo pior pois gastam mais tempo armazenando resultados do que economizam ao reusá-los.

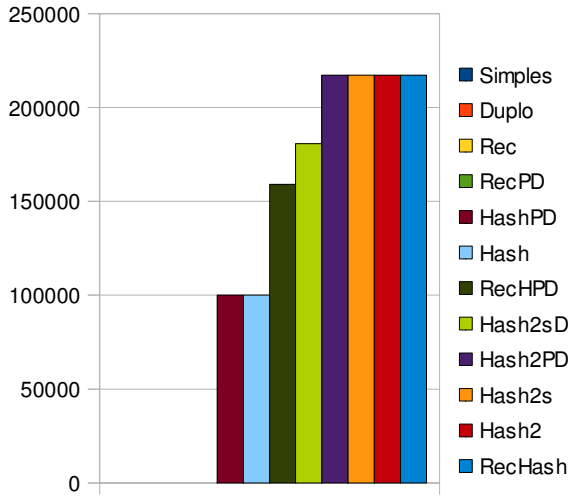

Figura 3 – soluções armazenadas nos *hashs*

A Figura 4 compara o desempenho das quatro soluções que não utilizam *hash* para contrastar as implementações iterativas e a recursivas. É importante notar que as soluções recursivas têm pior desempenho do que as equivalentes iterativas. É importante lembrar que soluções recursivas tendem a estourar a pilha de execução (*heap*) e por isso não poderiam ser utilizadas para intervalos muito grandes.

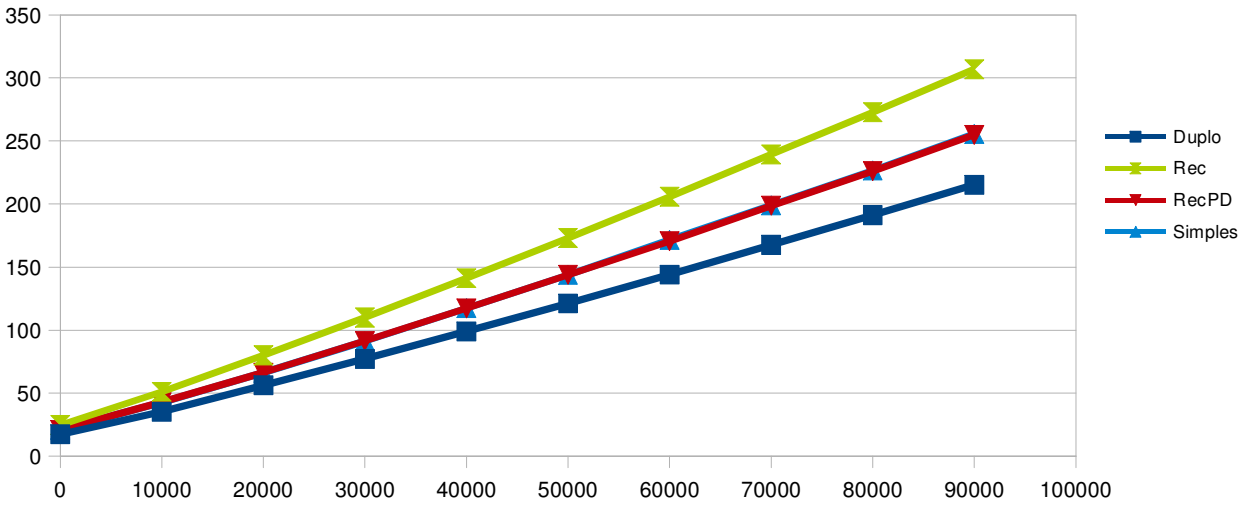

Figura 4 – comparação entre soluções iterativas e recursivas

A Figura 5 apresenta a comparação entre duas implementações que usam *hashs* e armazenam soluções parciais: *Hash2* e *Hash2s* a diferença entre elas é que a primeira utiliza um ***ArrayList*** e a segunda utiliza um **arranjo**. Nesta figura, pode-se notar que o uso do arranjo é mais eficiente do que o uso da classe *ArrayList*, porém é importante lembrar que o arranjo precisa ter toda a memória alocada inicialmente (o que pode representar um desperdício de memória ou um limitante no intervalo de números que o algoritmo poderá executar).

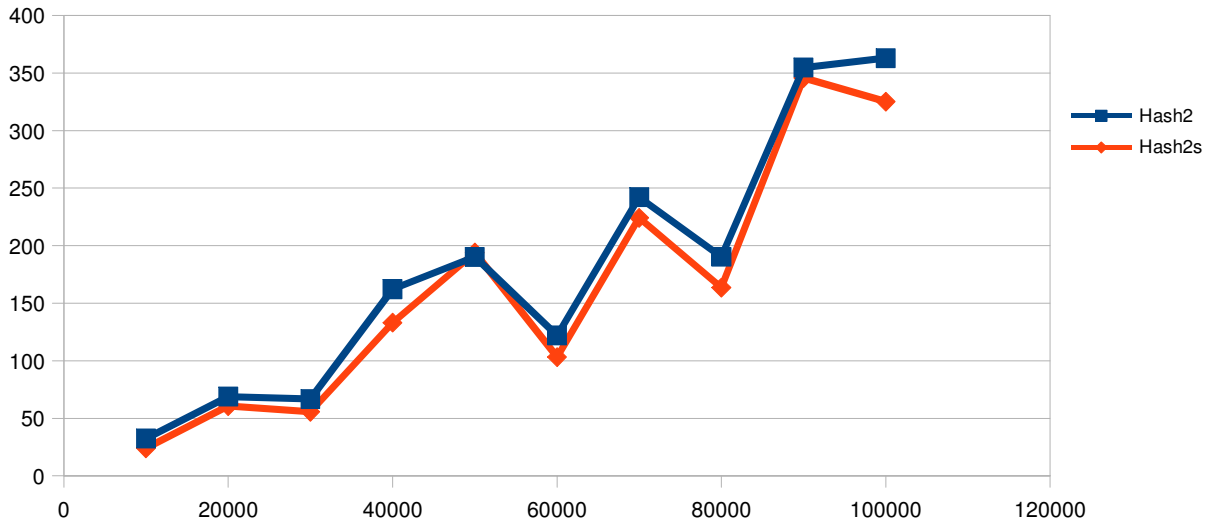

Figura 4 – comparação entre solução que usa e que não usa um *ArrayList*

Testes com intervalos maiores foram feitos utilizando apenas quatro implementações: *Duplo*, *Hash*, *HahPD* e *Simples*. As outras implementações não executariam devido ao excesso de uso de memória ou recursões muito profundas.

A Figura 5 apresenta a comparação de desempenho das implementações considerando intervalos de 100.000 números. Note que ao se utilizar esses intervalos (por exemplo, de 300.001 a 400.000) ao invés de todos os números de 1 até o limite superior do intervalo (por exemplo, de 1 a 400.000) este teste não favorece o uso de *hash* (que apresenta melhores resultados quanto maior a quantidade de números processados no mesmo teste).

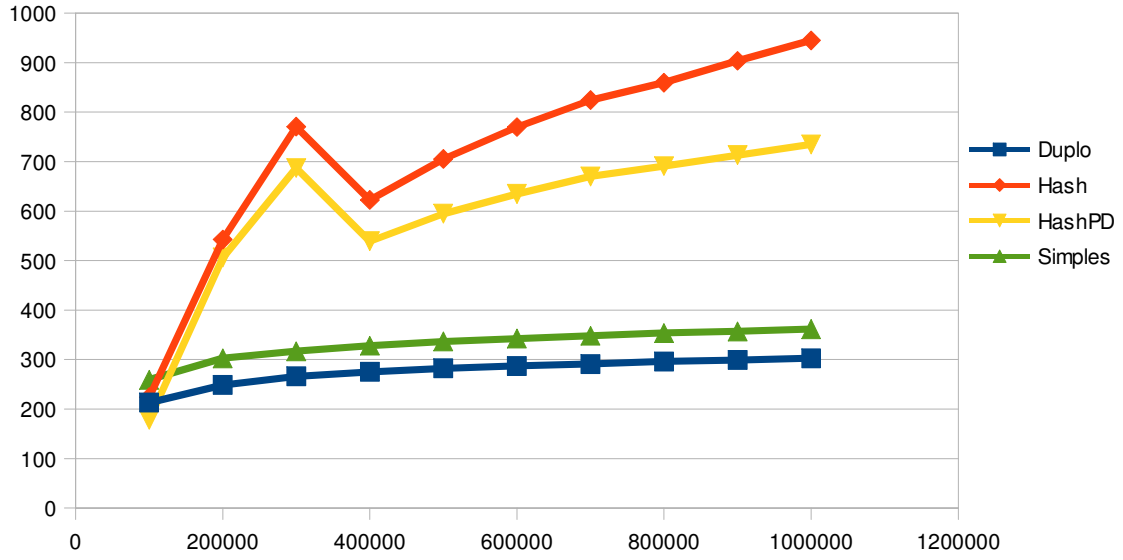

Figura 5 – comparação de desempenho utilizando intervalos de 100.000 números

A Figura 6 apresenta a comparação de desempenho das implementações considerando intervalos de 1 até os limites apresentados na figura. Este tipo de teste favorece o uso de *hashs*.

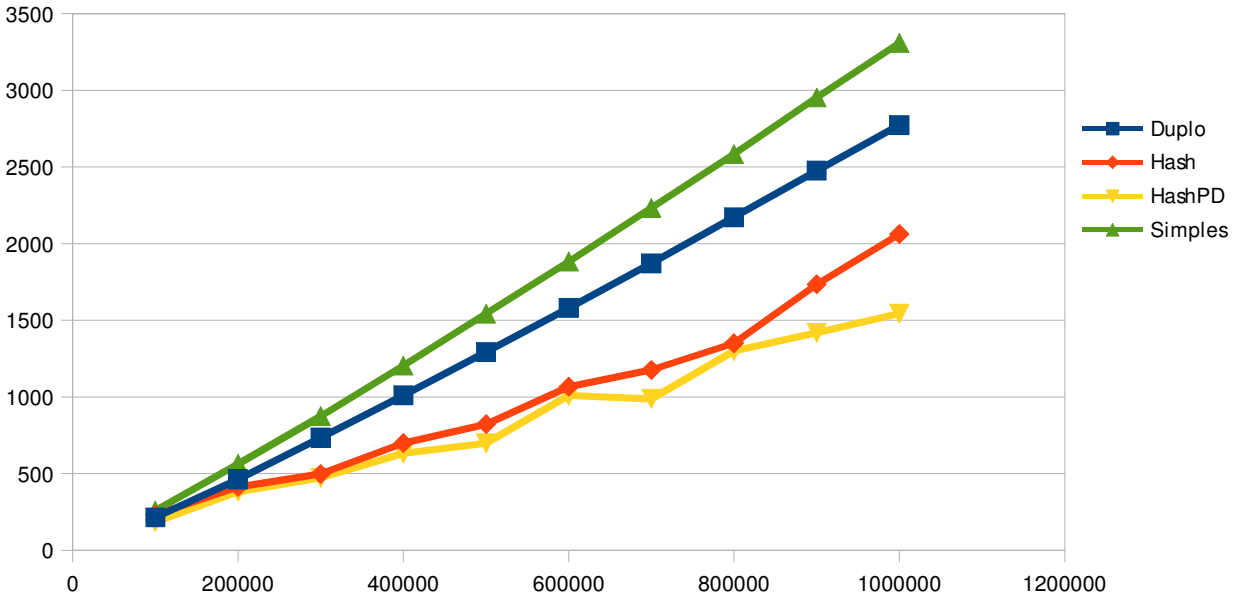

Figura 6 – comparação de desempenho utilizando valores de 1 a 1.000.000

Restrições das soluções:

As soluções recursivas possuem quanto a quantidade de chamadas recursivas devido ao tamanho da pilha de execução (*heap*).
As soluções que utilizam *hash* possuem como restrição o total de memória utilizada pelo *hash*.
Por fim, as únicas soluções com menores restrições são a *Simples* e a *Duplo*. A única restrição destas soluções é que o valor de n não pode ultrapassar o valor máximo de um *int* (inteiro de 16 bits). Esta restrição pode ser relaxada trocando-se *int* por *long* no código (mas ainda existirá uma restrição).

Conclusões

Para intervalos arbitrários e considerando as restrições apresentadas a melhor solução é a utilizada pela implementação "*Duplo*". Para intervalos de 1 até algum valor, a melhor solução é a que utiliza *hash* e "passo duplo": *HashPD.*