Exponenciação por sucessivas multiplicações

Queremos resolver a^n : exp(a, n)

Indução Fraca:

Caso base: para n = 0 temos que $a^0 = 1$

Passo indutivo: assumimos que sabemos resolver: **exp(a, n-1)**

O que falta para resolver exp(a,n)? exp(a,n) = exp(a, n-1) * a

```
static double exp(double a, int n){
    if (n == 0) return 1;
    else return a*exp(a,n-1);
}
```

T(n) = 0 para n = 0

T(n) = 1*T(n-1) + 1 para n > 0

Se resolvermos a equação teremos:

T(n) = n

T(n) ϵ Θ(n)

Indução Forte:

Caso base: para n = 0 temos que $a^0 = 1$

Passo indutivo: assumimos que sabemos resolver: **exp(a, n/2)**

O que falta para resolver exp(a,n)? exp(a,n) = exp(a, n/2) * exp(a, n/2)

Atenção: estamos trabalhando com valores de n inteiros então o código tem que tratar o fato de n ser par ou ímpar. Adicionalmente não faz sentido chamar duas vezes exp(a, n/2) com os mesmos parâmetros (iremos chamar uma única vez e multiplicar o resultado por ele mesmo.

Divisão: dividir n por 2 / **Conquista**: uma única chamada recursiva para exp(a, n/2) / **Combinação**: elevar ao quadrado (multiplicar pelo próprio valor) o resultado de exp(a, n/2) para n par e elevar ao quadro e multiplicar com a em caso de n ímpar.

```
static double exp2(int a, int n){
    if (n == 0) return 1;
    else{
        double aux = exp2(a, n/2);
        aux = aux * aux;
        if (n%2==1) aux = aux * a;
        return aux;
    }
}
```

Melhor caso:

T(n) = 0 para n = 0

T(n) = T(n/2) + 1 para n > 0 (par*)

Se resolvermos, teremos: **T(n) = log(n) para n>0**

Pior caso:

T(n) = 0 para n = 0

T(n) = T((n-1)/2) + 2 para n > 0 (ímpar**)

Se resolvermos, teremos: **T(n) = 2*$\lfloor \log(n) \rfloor$ + 2 para n>0**

Para ambos os casos **T(n) ϵ Θ(log n)**

* **para sempre ocorrer n precisa ser potência de 2**

** **para sempre ocorrer, n precisa ser potência de 2 menos 1**

Encontrar o maior valor em um arranjo

Indução Fraca:

Caso base: para n = 1 o único elemento "a ser olhado" no arranjo é o maior

Passo indutivo: assumimos que sabemos resolver: **maior(A, n-1)**

O que falta para resolver maior(A, n)? Comparar o n-ésimo elemento com o maior entre os n-1 elementos.

```
static int maior(int[] A, int n){
    if (n == 1) return A[0];
    else{
        int temp = maior(A, n-1);
        if (A[n-1] > temp) return A[n-1];
        else return temp;
    }
}
```

T(n) = 0 para n = 1

T(n) = 1*T(n-1) + 1 para n > 0

Se resolvermos a equação teremos:

T(n) = n-1

T(n) ϵ Θ(n)

Indução Forte:

Caso base: para n = 1 o único elemento "a ser olhado" no arranjo é o maior

Passo indutivo: assumimos que sabemos resolver os subproblemas:

maior(A, ini, meio) e **maior(A, meio+1, fim)**

O que falta para resolver maior(A, ini, fim)? Comparar a solução do primeiro subproblema com a do segundo e retornar a maior delas.

Divisão: as duas metades do arranjo (encontrar valor da posição do meio) / **Conquista**: uma chamada recursiva para cada metade / **Combinação**: comparar o maior elemento de cada metade e, assim, achar o maior global.

```
static int maior2(int[] A, int ini, int
fim){
    if (ini==fim) return A[ini];
    int meio = (fim+ini)/2;
    int t1 = maior2(A,ini,meio);
    int t2 = maior2(A, meio+1,fim);
    if (t1>t2) return t1;
    return t2;
}
```

Sendo n = fim - ini + 1

T(n) = 0 para n = 1

T(n) = 2*T(n/2) + 1 para n > 0 (n par)

T(n) = 1*T($\lceil n/2 \rceil$) + 1*T($\lfloor n/2 \rfloor$)+ 1 para n > 0 (n ímpar)

Se resolvermos a equação teremos:

T(n) = n-1

T(n) ϵ Θ(n)

Encontrar o menor e o maior valor em um arranjo

Indução Fraca:

Caso base: para n = 1 o único elemento "a ser olhado" no arranjo é o menor e o maior

Passo indutivo: assumimos que sabemos resolver: **menorMaior(A, n-1)**

O que falta para resolver menorMaior(A, n)? Comparar o n-ésimo elemento com o menor entre os n-1 elementos, se ele não for menor, comparar o maior dos n-1 elementos.

```
static int[] menorMaior(int[] A, int n){
    if (n==1) {
        int[] res = new int[2];
        res[0] = A[0];
        res[1] = A[0];
        return res;
    }else{
        int[] temp = menorMaior(A, n-1);
        if (A[n-1] < temp[0]) temp[0]=A[n-1];
        else if (A[n-1] > temp[1])
                                temp[1] = A[n-1];
        return temp;
    }
}
```

Melhor caso (o atual é sempre menor do que os anteriores)

T(n) = 0 para n = 1

T(n) = 1*T(n-1) + 1 para n > 0

Se resolvermos, teremos: **T(n) = n-1**

Pior caso (o atual nunca é menor do que o menor anterior [o primeiro número era o menor])

T(n) = 0 para n = 1

T(n) = 1*T(n-1) + 2 para n > 0

Se resolvermos, teremos: **T(n) = 2*n -2**

Para ambos os casos: **T(n) ϵ Θ(n)**

Indução Forte:

Caso base: para n = 1 o único elemento "a ser olhado" no arranjo é o maior

Passo indutivo: assumimos que sabemos resolver os subproblemas:

maior(A, ini, meio) e **maior(A, meio+1, fim)**

O que falta para resolver maior(A, ini, fim)? Comparar os elementos das soluções do primeiro subproblema com as do segundo e retornar a menor e a maior delas.

Divisão: as duas metades do arranjo (encontrar valor da posição do meio) / **Conquista**: uma chamada recursiva para cada metade / **Combinação**: comparar o menor elemento de cada metade e o maior elemento de cada metade e, assim, achar o menor e o maior globais.

```
static int[] menorMaior2(int[] A, int ini, int fim){
    if (ini==fim) {
        int[] res = new int[2];
        res[0] = A[ini];
        res[1] = A[ini];
        return res;
    }else{
        int meio = (fim+ini)/2;
        int[] t1 = menorMaior2(A,ini,meio);
        int[] t2 = menorMaior2(A, meio+1,fim);
        if (t1[0]>t2[0]) t1[0] = t2[0];
        if (t1[1]<t2[1]) t1[1] = t2[1];
        return t1;
    }
}
```

Sendo n = fim - ini + 1

T(n) = 0 para n = 1

T(n) = 2*T(n/2) + 2 para n > 0 (n par)

T(n) = 1*T($\lceil n/2 \rceil$) + 1*T($\lfloor n/2 \rfloor$) + 2 para n > 0 (n ímpar)

Se resolvermos, teremos:

T(n) = 2*n -2

T(n) ϵ Θ(n)