Algoritmos e Complexidade

Fabio Gagliardi Cozman

PMR2300
Escola Politécnica da Universidade de São Paulo

Algoritmos

- Um algoritmo é um procedimento descrito passo a passo para resolução de um problema em tempo finito.
- Formalização: máquinas de Turing.
- Algoritmos são julgados com base em vários fatores:
  - tempo de escrita;
  - complexidade de manutenção;
  - consumo de memória;
  - eficiência de execução.

Eficiência

Quanto à eficiência, vários fatores se inter-relacionam:

- qualidade do código;
- tipo do processador;
- qualidade do compilador;
- linguagem de programação.

Máximo valor

Conside um exemplo onde queremos encontrar o máximo valor em um arranjo de inteiros. Um algoritmo simples é varrer o arranjo, verificando se cada elemento é o maior até o momento (e caso seja, armazenando esse elemento). A seguinte função implementa esse algoritmo:

```
int arrayMax(int a[]) {
        int currentMax=a[0];
        for (int i=1; i<a.length; i++) {
                if (currentMax < a[i])
                        currentMax=a[i];
        }
        return(currentMax);
}
```

Complexidade

- Consideremos um exemplo no qual temos dois métodos para resolver o mesmo problema, cada um dos quais com uma complexidade diferente.
- Suponha que tenhamos um arranjo `x` e que queiramos calcular outro arranjo `a` tal que:

$$a[i] = \frac{\sum_{j=0}^{i} x[j]}{i+1}.$$

Complexidade

Uma solução é:

```
double[] medias(double x[]) {
        double a[]=new double[x.length];
        for (int i=0; i<x.length; i++) {
            double temp=0.0;
            for (int j=0; j<=i; j++)
                temp=temp+x[j];
            a[i]=temp/((double)(i+1));
        }
        return(a);
}
```

Complexidade

Qual é o custo desta função?
Note que o loop interno roda n vezes, onde n é o tamanho de `x`. Em cada uma dessas vezes, temos um número de operações proporcional a

$$1 + 2 + 3 + \cdots + (n-1) + n,$$

ou seja,

$$\sum_{i=0}^{n-1}(i+1) = \frac{n(n+1)}{2} = \frac{n^2+n}{2}.$$

Portanto o custo total deverá ser aproximadamente quadrático em relação ao tamanho de `x`.

Complexidade

Uma outra solução para o mesmo problema é:

```
double[] mediasLinear(int x[]) {
        double a[]=new double[x.length];
        double temp=0.0;
        for (int i=0; i<x.length; i++) {
                temp=temp+x[i];
                a[i]=temp/((double)(i+1));
        }
        return(a);
}
```

Essa solução envolve basicamente n operações (há um custo fixo e um custo para cada elemento de x); o custo total é proporcional a n.

Ordenação

- Ordenação é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma ordem predefinida.
- A ordenação de sequências é um dos mais importantes problemas em computação, dado o enorme número de aplicações que a empregam.

Ordenação por Seleção (em ordem decrescente)

```
public static void ordena(int a[]) {
    for (int i=0; i<a.length; i++) {
        int jmax=i;
        int max =a[i];
        for (int j=i+1; j<a.length; j++) {
            if (a[j]>max) {
                max=a[j];
                jmax=j;
            }
        }
        int temp=a[i];
        a[i]=max;
        a[jmax]=temp;
    }
}
```

Ordenação por Seleção

A complexidade desse algoritmo é quadrática no tamanho n de `a`, pois é proporcional a

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)n}{2}=\frac{n^2-n}{2}.$$

Ordenação por Inserção (em ordem crescente)

```
public static void ordena (int a[]) {
    for (int i=1; i<a.length; i++) {
        int temp=a[i];
        int j;
        for (j=1; (j>0)&&(temp<a[j-1]); j--)
            a[j]=a[j-1];
        a[j]=temp;
    }
}
```

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

$$\left\{\begin{array}{l} 6 \mid 3\ 4\ 5\ 9\ 8 \\ 3\ 6 \mid 4\ 5\ 9\ 8 \\ 3\ 4\ 6 \mid 5\ 9\ 8 \\ 3\ 4\ 5\ 6 \mid 9\ 8 \\ 3\ 4\ 5\ 6\ 9 \mid 8 \\ 3\ 4\ 5\ 6\ 8\ 9 \end{array}\right.$$

Ordenação por Inserção

- Note que a complexidade de ordenação por inserção varia com a entrada!
- O pior caso é aquele em que o arranjo está ordenado em ordem decrescente.
- O melhor caso, aquele em que o arranjo já está ordenado em ordem crescente.
- No *melhor caso*, o custo é proporcional a n.
- No *pior caso*, o custo é proporcional a

$$1 + 2 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}.$$

Definição: análise assintótica

- Para quantificar a complexidade de um algoritmo, vamos usar a ordem de crescimento do tempo de processamento em função do tamanho da entrada.
- Vamos assumir que todo algoritmo tem uma única entrada crítica cujo tamanho é N (por exemplo, o comprimento do arranjo a ser ordenado).
- A notação para indicar a ordem de um algoritmo é denominada "**BigOh**". Temos:
  - $\mathcal{O}(N)$: ordem linear;
  - $\mathcal{O}(N^2)$: ordem quadrática;
  - $\mathcal{O}(2^N)$: ordem exponencial;
  - $\mathcal{O}(\log N)$: ordem logarítmica.

Definição formal

- O custo $T(N)$ de um algoritmo com entrada de tamanho N é $\mathcal{O}(f(N))$ se existem constantes $K > 1$ e M tal que:

$$T(N) \leq K \cdot f(N), \qquad \forall N \geq M.$$

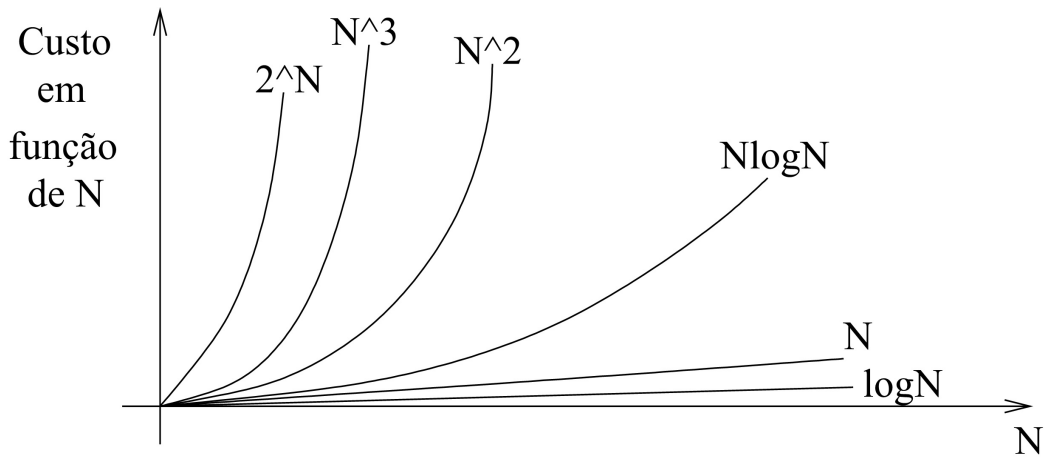
- Ou seja, se um algoritmo é $\mathcal{O}(f(N))$, então há um ponto M a partir do qual o desempenho do algoritmo é limitado a um múltiplo de $f(N)$.

Consequências da definição

- $\mathcal{O}(N^3) + \mathcal{O}(N^2) = \mathcal{O}(N^3)$;
- $\mathcal{O}(N^2) + N = \mathcal{O}(N^2)$;
- $\mathcal{O}(\sum_{i=1}^{k} a_i N^i) = \mathcal{O}(N^k)$.
- $\mathcal{O}(1)$ indica um trecho de programa com custo constante.

Crescimento do esforço computacional

As diversas ordens de algoritmos podem ser esquematizadas como segue:



Observação 1

- Um algoritmo A_1 pode ser mais rápido que outro algoritmo A_2 para pequenos valores de N, e no entanto ser pior para grandes valores de N.
- A análise por notação ”BigOh” se preocupa com o comportamento para *grandes* valores de N, e é por isso denominada *análise assintótica*

Observação 2

- Um algoritmo pode ter comportamentos diferentes para diferentes tipos de entradas; por exemplo, ordenação por inserção depende da entrada estar ordenada ou não.
- Em geral a complexidade é considerada *no pior caso*. Usaremos sempre essa convenção neste curso.

Observação 3

- A análise assintótica ignora o comportamento para N pequeno e ignora também custos de programação e manutenção do programa, mas esses fatores podem ser importantes em um problema prático.

Observação 4

- O custo de um algoritmo é sempre ”dominado” pelas suas partes com maior custo.
- Em geral, as partes de um algoritmo (ou programa) que consomem mais recursos são os laços, e é neles que a análise assintótica se concentra.
- Note que se dois ou mais laços se sucedem, aquele que tem maior custo ”domina” os demais.

Exemplos 1 e 2

- ```
  for (int i=0; i<n; i++)
      sum++;
  ```

  Custo é $\mathcal{O}(N)$.

- ```
  for (int i=0; i<(n*n); i++)
      for (int j = i; j<n; j++)
          sum++;
  ```

  Custo é $\mathcal{O}(N^2)$.

Exemplo 3

Consideremos um exemplo, a procura da subsequência máxima (o problema é encontrar uma subsequência de um arranjo de inteiros, tal que a soma de elementos dessa subsequência seja máxima). Por exemplo:

$$-2, \underbrace{11, -4, 13}_{\text{subseq máx}}, -5, 2,$$

Podemos codificar soluções cúbicas, quadráticas e lineares para esse problema.

Solução cúbica

```
static private int seqStart = 0; // best sequence
static private int seqEnd = 0;  // best sequence

public static int maxSubSum( int a[] ) {
  int maxSum = 0;
  for(int i = 0; i < a.length; i++)
     for(int j = i; j < a.length; j++) {
        int thisSum = Integer.MIN_VALUE;
        for(int k = i; k <= j; k++)
           thisSum += a[ k ];
        if( thisSum > maxSum ) {
           maxSum   = thisSum;
           seqStart = i;     seqEnd   = j;
        }
     }
  return(maxSum);
}
```

Análise

Essa solução tem custo:

$$\mathcal{O}\left(\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j}1\right) = \mathcal{O}\left(\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}(j-i+1)\right)$$

$$= \mathcal{O}\left(\sum_{i=0}^{N-1}\frac{(N-i)(N+i-1)}{2} + i(i-N) + (N-i)\right)$$

$$= \mathcal{O}(N^3)$$

Somas

Para fazer esse tipo de análise, é importante nos lembrarmos de algumas fórmulas:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \qquad \sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} \qquad \sum_{i=0}^{N-1} i^2 = \frac{N^3}{3} - \frac{N^2}{2} + \frac{N}{6}$$

$$\sum_{i=1}^{N} i^3 = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4} \qquad \sum_{i=0}^{N-1} i^3 = \frac{N^4}{4} - \frac{N^3}{2} + \frac{N^2}{4}$$

Solução quadrática

```
static private int seqStart = 0;
static private int seqEnd = 0;

public static int maxSubSum(int a[]) {
    int maxSum = 0;
    for (int i=0; i < a.length; i++) {
        int thisSum = 0;
        for (int j=i; j<a.length; j++ ) {
            thisSum += a[j];
            if (thisSum > maxSum) {
                maxSum = thisSum;
                seqStart = i;   seqEnd   = j;
            }
        }
    }
    return maxSum;
}
```

Resumindo:

1. Saiba avaliar a complexidade e identificar pontos críticos, principalmente focando em laços;
2. Concentre otimizações em pontos críticos, somente após se certificar que o algoritmo funciona.

Complexidade logarítmica

- Vimos até agora exemplos de complexidade polinomial, ou seja, $\mathcal{O}(N^{\alpha})$.
- Um tipo importante de algoritmo é o que tem complexidade logarítmica, ou seja, $\mathcal{O}(\log N)$.
- IMPORTANTE: a base do logaritmo não importa, pois

$$\mathcal{O}(\log_{\alpha} N) = \mathcal{O}\left(\frac{\log_{\beta} N}{\log_{\beta} \alpha}\right) = \mathcal{O}(\log_{\beta} N).$$

Exemplo 1

- Considere que uma variável x é inicializada com 1 e depois é multiplicada por 2 um certo número K de vezes.
- Qual é o número K^* tal que x é maior ou igual a N?
- Esse problema pode ser entendido como uma análise do seguinte laço:

```
int x=1;
while (x<N) {
        ...
        x=2*x;
        ...
}
```

Análise

- A questão é quantas iterações serão realizadas.
- Note que se o interior do laço tem custo constante $\mathcal{O}(1)$, então o custo total é $\mathcal{O}(K^*)$.
- A solução é simples: queremos encontrar K tal que:

$$2^K \geq N \Rightarrow K \geq \log_2 N,$$

e portanto $K^* = \lceil \log N \rceil$ garante que $\mathtt{x}$ é maior ou igual a N.

Exemplo 2

- Considere agora que uma variável x é inicializada com N e depois é dividida por 2 um certo número K de vezes.
- Qual é o número K^* tal que x é menor ou igual a 1?
- De novo podemos entender esse problema como o cálculo do número de iterações de um laço:

```
int x=N;
while (x>1) {
            ...
            x=x/2;
            ...
}
```

Análise

- A solução é dada por:

$$N\left(\frac{1}{2}\right)^K \leq 1 \Rightarrow N \leq 2^K \Rightarrow 2^K \geq N \Rightarrow K \geq \log_2 N.$$

- De novo, temos que $K^* = \lceil \log N \rceil$ é a solução.

Logaritmos

- Nessas duas situações a complexidade total é $\mathcal{O}(\lceil \log N \rceil)$, supondo que o custo de cada iteração é constante.
- Note que $\lceil \log N \rceil \leq \log N + 1$ e portanto:

$$\mathcal{O}(\lceil \log N \rceil) = \mathcal{O}(\log N + 1) = \mathcal{O}(\log N).$$

Busca não-informada

- Considere um arranjo de tamanho N e suponha que queremos encontrar o índice de um elemento $\mathtt{x}$.
- O algoritmo de *busca sequencial* simplesmente varre o arranjo do início ao fim, até encontrar o elemento procurado.

Busca sequencial

```
int buscaSequencial(int a[], int x) {
        for (int i=0; i<a.length; i++)
                if (a[i]==x)
                        return(i);
}
```

Custo

- O custo desse algoritmo no pior caso é $\mathcal{O}(N)$.
- Em média, podemos imaginar que $\mathtt{x}$ está em qualquer posição com igual probabilidade, e temos um custo médio proporcional a $\frac{N}{2}$, ainda de ordem $\mathcal{O}(N)$.

Busca binária

- Suponha agora que o arranjo está ordenado. Nesse caso podemos dividir o problema em dois a cada passo, verificando se o elemento está na metade superior ou inferior do arranjo em consideração.

Algoritmo

```
int buscaBinaria(int a[], int x) {
    int low=0;
    int high=a.length-1;
    int mid;
    while (low<=high) {
        mid=(low + high)/2;
        if (a[mid]<x)
            low=mid+1;
        else {
            if (a[mid]>x)
                high=mid-1;
            else
                return(mid);
        }
    }
    return(-1); // Nao encontrou
}
```

Custo

- Esse programa faz uma iteração que recebe um arranjo de tamanho N, depois $\frac{N}{2}$, depois $\frac{N}{4}$, e assim sucessivamente; no pior caso isso prossegue até que o tamanho seja 1.
- Portanto o número de iterações é $\mathcal{O}(\lceil \log N \rceil)$ e o custo total é $\mathcal{O}(\log N)$, já que o custo de cada iteração é constante.

Recursão

- Um procedimento recursivo é um procedimento que chama a si mesmo durante a execução.
- Recursão é uma técnica importante e poderosa; algoritmos recursivos devem ter sua complexidade assintótica analisada com cuidado.

Definição

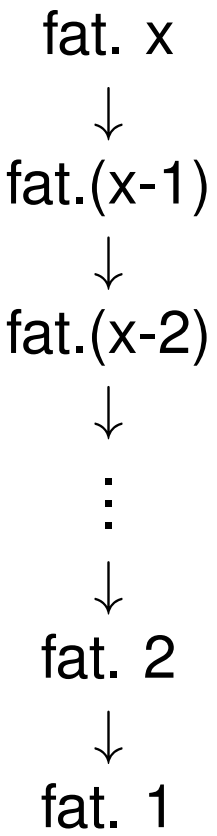
Considere como exemplo um programa que calcula fatorial:

```
long fatorial(int x) {
        if (x<=1)
                return(1);
        else
                return(x*fatorial(x-1));
}
```

Análise

- O método funciona para $x = 1$; além disso, funciona para $x \geq 1$ se funciona para $(x - 1)$. Por indução finita, o método calcula os fatoriais corretamente.
- O caso $x = 1$, em que ocorre a parada do algoritmo, é chamado *caso base* da recursão.

Árvore de recursão



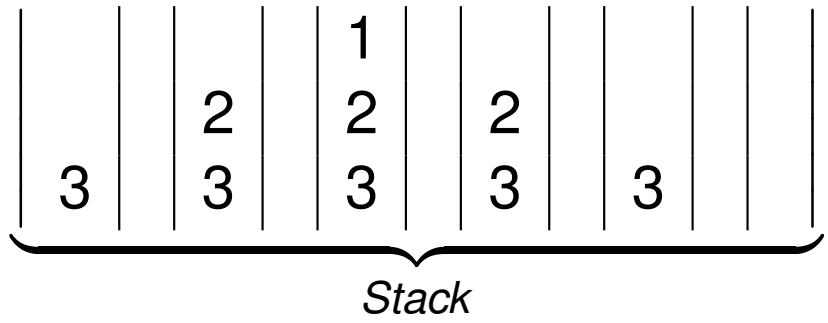
Busca binária recursiva

```
int buscaBinaria(int a[],int x,int low,int high) {
  if (low>high)
        return(-1);
  int mid=(low+high)/2;
  if (a[mid]<x)
        return(buscaBinaria(a,x,(mid+1),high));
  else
       if (a[mid]>x)
            return(buscaBinaria(a,x,low,(mid-1)));
       else
            return(mid);
}
```

Controle via Stack

- Um procedimento recursivo cria várias “cópias” de suas suas variáveis locais no Stack à medida que suas chamadas são feitas.
- Sempre é necessário avaliar a complexidade de um procedimento recursivo com cuidado, pois um procedimento recursivo pode “esconder” um laço bastante complexo.
- Consideremos a função recursiva fatorial. Uma execução típica seria:

$$\text{fatorial}(3) \rightarrow \text{fatorial}(2) \rightarrow \text{fatorial}(1)$$



Custo de fatorial

O custo para uma chamada `fatorial(N)` é $\mathcal{O}(N)$. Uma solução iterativa equivalente seria:

```
long fatorial(int N) {
        long fat=1;
        for (int i=2; i<=N; i++)
                fat=fat*i;
        return(fat);
}
```

Essa solução tem claramente custo $\mathcal{O}(N)$.

Mergesort

- Consideremos o algoritmo de *ordenação por união (mergesort)*.
- Considere um arranjo `a` de tamanho N. Para ordená-lo, divida o arranjo em 2 partes, ordene cada uma e una as duas partes (com custo $\mathcal{O}(N)$). O algoritmo completo é mostado a seguir.

Algoritmo - parte 1

```
void mergesort(int a[]) {
    int temp[]=new int[a.length];
    mergesort(a, temp, 0, a.length);
}
```

Algoritmo - parte 2

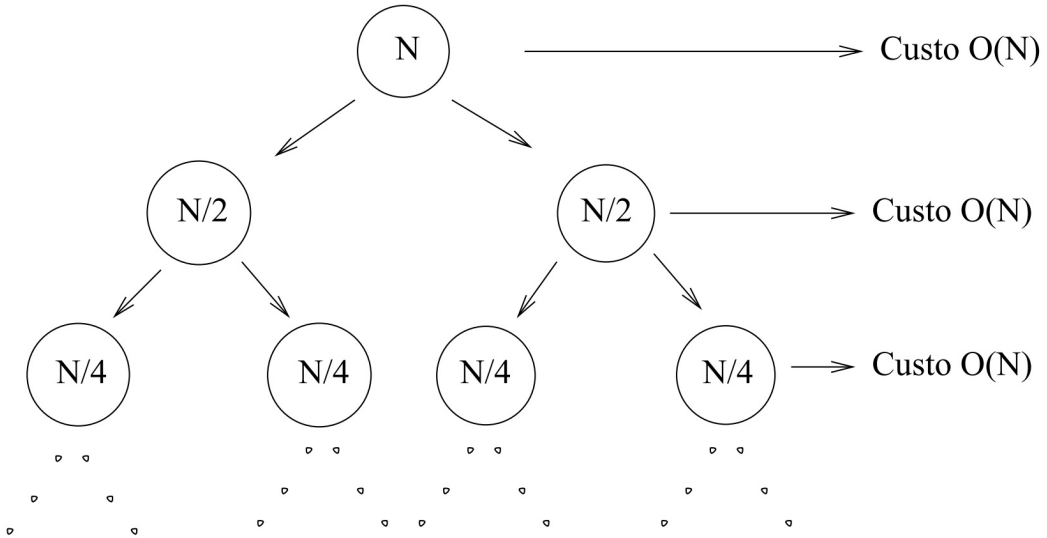
```
void mergesort(int a[], int temp[],
        int left, int right) {
        if (left<right) {
            int center=(left+right)/2;
            mergesort(a, temp, left, center);
            mergesort(a, temp, (center+1), right);
            merge(a, temp, left, center, (center+1
        }
}
```

Algoritmo - parte 3

```
void merge(int a[], int temp[],
        int leftStart, int leftEnd,
        int rightStart, int rightEnd) {
        int start=leftStart;
        int aux=start;
        while ((leftStart<=leftEnd) &&
                    (rightStart<=rightEnd)) {
            if (a[leftStart]<a[rightStart])
                    temp[aux++]=a[leftStart++];
            else
                    temp[aux++]=a[rightStart++];
        }
        while (leftStart<=leftEnd)
            temp[aux++]=a[lefStart++];
        while (rightStart<=rightEnd)
            temp[aux++]=a[rightStart++];
        for (int i=start; i<=rightEnd; i++)
            a[i]=temp[i];
}
```

Custo

Com um arranjo de tamanho *N*, temos a seguinte árvore de recursão (estamos assumindo que *N* é uma potência de 2):



Relação de recorrência

O custo em cada nó, $\mathcal{O}(N)$, é o custo da combinação das soluções. Suponha que N seja uma potência de 2. Nesse caso teremos $\log_2 N$ níveis, cada um com custo $\mathcal{O}(N)$. O custo total é $\mathcal{O}(N \log N)$.
Uma maneira geral de calcular esse custo é considerar que o custo total $T(N)$ é regulado pela seguinte relação de recorrência:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + C \cdot N.$$

Solução

Podemos escrever:

$$\begin{aligned} T\left(\frac{N}{2}\right) &= 2 \cdot T\left(\frac{N}{4}\right) + C \cdot \left(\frac{N}{2}\right) \\ \Longrightarrow T(N) &= 2 \cdot \left(2 \cdot T\left(\frac{N}{4}\right) + C \cdot \left(\frac{N}{2}\right)\right) + C \cdot N \\ &= 4 \cdot T\left(\frac{N}{4}\right) + 2 \cdot C \cdot N. \end{aligned}$$

Seguindo esse raciocínio, chegamos a

$$T(N) = 2^K \cdot T\left(\frac{N}{2^K}\right) + K \cdot C \cdot N$$

para uma recursão de K níveis.

Solução

Sabemos que o número total de níveis é $\log_2 N$ (para N potência de 2),

$$\begin{aligned} T(N) &= 2^{\log_2 N} \cdot T\left(\frac{N}{2^{\log_2 N}}\right) + C \cdot N \cdot \log_2 N \\ &= N \cdot T\left(\frac{N}{N}\right) + C \cdot N \cdot \log N \\ &= C' \cdot N + C \cdot N \cdot \log N = O(N \cdot \log N), \end{aligned}$$

pois temos $T(1) = \mathcal{O}(1)$ nesse algoritmo.

Solução

Se a suposição que N é uma potência de 2 for removida, teremos um custo de no máximo $\mathcal{O}(N)$ em cada nível, e um número máximo de $(\log_2 N + 1)$ níveis. Nesse pior caso,

$$\begin{aligned} T(N) &= 2^{\log_2 N+1} \cdot T\left(\frac{N}{2^{\log_2 N}}\right) + C \cdot N \cdot \log_2 N + 1 \\ &= 2 \cdot N \cdot T(1) + C \cdot N \cdot \log N + C \cdot N \\ &= O(N \log N), \end{aligned}$$

se considerarmos que $T(1)$ é uma constante (já que esse custo nunca é realmente atingido, pois todas as recursões "param" quando $N = 1$).

Divisão e conquista

Existem muitos problemas que podem ser resolvidos pelo método genérico de "divisão-e-conquista", no qual o problema é dividido em partes que são resolvidas independentemente e depois combinadas. Esquematicamente temos:

- Problema original dividido em A sub-problemas, cada um com entrada N/B.
- Cada sub-problema dividido em A sub-problemas, cada um com entrada $(N/B)/B$.
- etc etc
- Até que cada sub-problema seja resolvido.

Custo

- Podemos em geral obter uma relação de recorrência:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + c \cdot f(N),$$

onde $f(N)$ é o custo de combinar os subproblemas.

- Um resultado geral é o seguinte: A relação $T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L$, com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Exemplo

- Suponha que uma recursão resolve a cada nível 3 subproblemas, cada um com metade do tamanho de chamada, e com custo linear para combinar subproblemas.
- Ou seja, $A = 3$, $B = 2$ e $L = 1$.
- Como $A > B^L$, sabemos que

$$T(N) = \mathcal{O}\left(N^{\log_2 3}\right) = \mathcal{O}(N^{1.59}).$$

Análise

- Supondo que N é potência de 2, temos no primeiro nível um custo maior ou igual que $c \cdot N$;
- no segundo nível um custo maior ou igual que $3 \cdot c \cdot \frac{N}{2}$;
- no terceiro nível um custo $\leq 3^2 \cdot c \cdot \frac{N}{2^2}$.

Análise detalhada

Com essas considerações, chegamos a

$$\begin{aligned} T(N) &= (\text{custo total das } 3^{\log_2 N} \text{folhas} = 3^{\log_2 N}) + \\ &\quad (c \cdot N + 3 \cdot c \cdot \frac{N}{2} + 3^2 \cdot c \cdot \frac{N}{2^2} + \cdots) \\ &= N^{\log_2 3} + c \cdot N \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i, \end{aligned}$$

onde k é o número de níveis da recursão, igual a $\log_2 N$.

Solução

$$
\begin{aligned}
T(N) &= N^{\log_2 3} + c \cdot N \cdot \sum_{i=0}^{\log_2 N - 1} \left(\frac{3}{2}\right)^i \\
&= N^{\log_2 3} + cN \frac{\left(\frac{3}{2}\right)^{\log_2 N} - 1}{\frac{3}{2} - 1} \\
&= N^{\log_2 3} + (cN)\left(2\frac{3^{\log_2 N}}{2^{\log_2 N}} - 2\right) \\
&= N^{\log_2 3} + \frac{cN2}{N} N^{\log_2 3} - 2cN \\
&= N^{\log_2 3} + 2cN^{\log_2 3} - 2cN \\
&= O\left(N^{\log_2 3}\right),
\end{aligned}
$$

onde usamos

$$
\sum_{i=0, a>0, a\neq 1}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}.
$$

Teorema

- $\mathcal{O}(N^a - N^b) = \mathcal{O}(N^a)$ se $a > b$.
  - Prova: $cN^a \geq c(N^a - N^b)$.
  - Além disso, $N^{a-\epsilon} < N^a - N^b$ para qualquer $\epsilon > 0$, quando N cresce.
    (Suponha $N^a/N^\epsilon \geq (N^a - N^b)$; então $N^\epsilon(1 - N^{b-a}) \leq 1$, o que é impossível para N grande.)

N qualquer

Essa solução assume que N é potência de 2; se isso não ocorre, o número de níveis é menor que $(\log_2 N + 1)$ e a complexidade ainda é $\mathcal{O}\left(N^{\log_2 3}\right)$.

Resumindo:

1. dado um programa recursivo, determine a relação de recorrência que rege o programa;
2. substitua os custos assintóticos em notação $\mathcal{O}(\cdot)$ por funções;
3. obtenha expressões para $T(N)$, resolvendo somatórios ou produtórios.

Resultado geral

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L, \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L, \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L. \end{cases}$$

Temos:

$$T(N) = cN^L + Ac\frac{N^L}{B^L} + A^2c\frac{N^L}{B^{2L}} + \cdots$$

Análise detalhada

$$
\begin{aligned}
T(N) &= AT(N/B) + cN^L \\
&= A(AT(N/B^2) + c(N/B)^L) + cN^L \\
&= A(A(AT(N/B^3) + c(N/B^2)^L) + c(N/B)^L) + cN^L \\
&= A^3 T(N/B^3) + A^2 c(N/B^2)^L + Ac(N/B)^L + cN^L \\
&= A^3 T(N/B^3) + cN^L(1 + A/B^L + (A/B^L)^2).
\end{aligned}
$$

Portanto, para k níveis:

$$
\begin{aligned}
T(N) &= A^k T(N/B^k) + cN^L(1 + A/B^L + \cdots + (A/B^L)^{k-1}) \\
&= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} (A/B^L)^i.
\end{aligned}
$$

Análise detalhada: Caso 1

Se $A/B^L = 1$, temos:

$$\begin{aligned} T(N) &= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} 1 \\ &= A^k T(N/B^k) + cN^L k. \end{aligned}$$

Tomando $k = \log_B N$, temos:

$$\begin{aligned} T(N) &= A^{\log_B N} T(N/B^{\log_B N}) + cN^L \log_B N \\ &= N^{\log_B A} T(N/N) + cN^L \log_B N \\ &= N^{\log_B A} c' + cN^L \log_B N \\ &= c' N^L + cN^L \log_B N \end{aligned}$$

pois $\log_B A = L$, e portanto obtemos $\mathcal{O}(N^L \log N)$.

Análise detalhada: Caso 2

Se $A/B^L < 1$, temos:

$$\begin{aligned} T(N) &= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} (A/B^L)^i \\ &= A^k T(N/B^k) + cN^L \left(\frac{(A/B^L)^k - 1}{A/B^L - 1} \right). \end{aligned}$$

Tomando $k = \log_B N$, temos:

$$\begin{aligned} T(N) &= A^{\log_B N} T(N/B^{\log_B N}) + cN^L \left(\frac{1 - (A/B^L)^{\log_B N}}{1 - A/B^L} \right) \\ &= N^{\log_B A} T(N/N) + c'' N^L (1 - (A/B^L)^{\log_B N}) \\ &= N^{\log_B A} c' + c'' N^L - c'' N^{\log_B A} \end{aligned}$$

e notando que $\log_B A < L$, obtemos $\mathcal{O}(N^L)$.

Análise detalhada: Caso 3

Se $A/B^L > 1$, temos:

$$\begin{aligned} T(N) &= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} (A/B^L)^i \\ &= A^k T(N/B^k) + cN^L \left(\frac{1 - (A/B^L)^k}{1 - A/B^L} \right). \end{aligned}$$

Tomando $k = \log_B N$, temos:

$$\begin{aligned} T(N) &= A^{\log_B N} T(N/B^{\log_B N}) + cN^L \left(\frac{(A/B^L)^{\log_B N} - 1}{A/B^L - 1} \right) \\ &= N^{\log_B A} T(N/N) + c'' N^L ((A/B^L)^{\log_B N} - 1) \\ &= N^{\log_B A} c' + c'' N^{\log_B A} - c'' N^L \end{aligned}$$

e notando que $\log_B A > L$, obtemos $\mathcal{O}(N^{\log_B A})$.

Exemplo: Inversão de Matrizes

- O algoritmo de Strassen para inversão de matrizes $N \times N$ divide o número de linhas pela metade, mas realiza sete chamadas recursivas por vez, com custo $\mathcal{O}(N^2)$ de combinação.
- Portanto o custo de inversão de uma matriz é $\mathcal{O}(N^{\log_2 7})$.

Little oh

- $T(N)$ é $o(f(N))$ se existe M positivo tal que

$$T(N) \leq \epsilon |f(N)|$$

  para todo $N \geq M$ e todo $\epsilon > 0$.
- Isto é, $\lim_{x \to \infty} f(x)/g(x) = 0$.
- Little oh não é muito usado (não vamos usar nesse curso).

Big Omega e Big Theta

- $T(N)$ é $\Omega(f(N))$ se existem k e M positivos tal que

$$T(N) \geq kf(N)$$

para todo $N \geq M$.

- $T(N)$ é $\Theta(f(N))$ se existem k_1, k_2 e M positivos tal que

$$k_1 f(N) \leq T(N) \leq k_2 f(N)$$

para todo $N \geq M$.

Exercício

Prove: se $f(N)$ é $\Theta(N^L)$, então

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + f(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L, \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L, \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L. \end{cases}$$

Formulário

- $b^{\log_c a} = a^{\log_c b}$;
- $\log_b a = \frac{\log_c a}{\log_c b}$;
- $\sum_{i=1}^{n} f(i) - f(i-1) = f(n) - f(0)$;
- $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$;
- $\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$;
- $\sum_{i=1}^{N} i^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6}$;
- $\sum_{i=0}^{N-1} i^2 = \frac{N^3}{3} - \frac{N^2}{2} + \frac{N}{6}$;
- $\sum_{i=1}^{N} i^3 = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4}$;
- $\sum_{i=0}^{N-1} i^3 = \frac{N^4}{4} - \frac{N^3}{2} + \frac{N^2}{4}$;
- $\sum_{i=0,a>0,a\neq 1}^{k} a^i = \frac{a^{k+1}-1}{a-1}$
- $\sum_{i=0,a\in[0,1]}^{\infty} a^i = \frac{1}{1-a}$
- $\sum_{i=0,a>0,a\neq 1}^{n} i \cdot a^i = \frac{a-(n+1)a^{n+1}+na^{n+2}}{(1-a)^2}$

Análise no caso médio: quicksort

- O algoritmo mais usado para ordenação é o quicksort, que tem pior caso $\mathcal{O}(N^2)$ e caso médio $\mathcal{O}(N \log N)$, além de não exigir armazenamento adicional.
- Na prática o quicksort é muito rápido para arranjos não ordenados.

Algoritmo

```
void quicksort(int s[], int a, int b) {
        if (a>=b) return;
        int p=s[b];      // pivot
        int l=a;
        int r=b-1;
        while (l<=r) {
            while ((l<=r)&&(s[l]<=p)) l++;
            while ((l<=r)&&(s[l]>=p)) r--;
                if (l<r) {
                    int temp=s[l];
                    s[l]=s[r];
                    s[r]=temp;
                }
        }
        s[b]=s[l];
        s[l]=p;
        quicksort(s, a, (l-1));
        quicksort(s, (l+1), b);
}
```