Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 9
Análise de Complexidade de Algoritmos

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Reis, Rocha e Faria, 2007)

Escola de Engenharia
Universidade do Minho

Introdução

- **Algoritmo: conjunto claramente especificado de instruções a seguir para resolver um problema**
- **Análise de algoritmos:**
  - Provar que um algoritmo está correto
  - Analisar e determinar os recursos exigidos por um algoritmo (em termos de tempo, espaço, etc.)
    - comparar os recursos exigidos por diferentes algoritmos que resolvem o mesmo problema (um algoritmo mais eficiente exige menos recursos para resolver o mesmo problema)
    - prever o crescimento dos recursos exigidos por um algoritmo à medida que o tamanho dos dados de entrada cresce
- **Soluções simples e fáceis de implementar nem sempre são as mais eficientes e até exequíveis!**

Complexidade Espacial e Temporal

- **Complexidade espacial de um programa ou algoritmo: espaço de memória que necessita para executar até ao fim**

  S(n) - espaço de memória exigido em função do tamanho (n) da entrada
- **Complexidade temporal de um programa ou algoritmo: tempo que demora a executar (tempo de execução)**

  T(n) - tempo de execução em função do tamanho (n) da entrada
- **Complexidade ↑ versus Eficiência ↓**
- **Por vezes estima-se a complexidade para:**
  - o "melhor caso" (pouco útil)
  - o "pior caso" (mais útil)
  - o "caso médio" (igualmente útil)

Notação de *O* grande

- **Na prática, é difícil (senão impossível) prever com rigor o tempo de execução de um algoritmo ou programa**
  - Para obter o tempo a menos de:
    - constantes multiplicativas (normalmente estas constantes são tempos de execução de operações atómicas)
    - parcelas menos significativas para valores grandes de *n*
  - Identificam-se as operações dominantes (mais frequentes ou muito mais demoradas) e determina-se o número de vezes que são executadas (e não o tempo de cada execução, que seria uma constante multiplicativa)
  - Exprime-se o resultado com a notação de *O* grande
  - Na prática um contador de operações inserido no programa ou a contagem do tempo são muito úteis para confirmar a complexidade.

Notação de *O* grande

- **Definição:**

  $T(n) = O(f(n))$ **(ler: *T(n)* é de ordem *f(n)*)**

  se e só se existem constantes positivas *c* e n_0 tal que $T(n) \leq cf(n)$ para todo o $n > n_0$

- **Exemplos:**

  $c_k n^k + c_{k-1} n^{k-1} + \ldots + c_0 = O(n^k)$ (c_i - constantes)

  $\log_2 n = O(\log n)$
  (não se indica a base porque mudar de base é multiplicar por constante)

  $4 = O(1)$ (usa-se 1 para ordem constante)

Análise de Algoritmos

- **Geralmente compromisso entre Espaço ocupado/Tempo de execução:**
  – Por exemplo para ser mais rápido usa-se estruturas de dados auxiliares
- **Estruturas de Dados e Algoritmos essencialmente úteis para problemas complexos. Para problemas simples qq serve:**
  – Por exemplo se o espaço de soluções for pequeno usa-se "gerar e testar"
- **Na prática (no futuro em projectos que envolvam programação):**
  – Se existe feito então usa-se(referindo a fonte ;-))
  – Senão, se existe parecido feito, então adapta-se (referindo a fonte ;-))
  – Senão desenvolve-se de raíz!
  – Adopta-se sempre a solução mais simples.
  – Não se complica desnecessariamente

Ordens de Crescimento

- **Classes de Crescimento da complexidade de algoritmos:**
  - O(1) : constante
  - O(log n) : logaritmico
  - O(n) : linear
  - O(nlog n) : n * log n
  - O(nk) : polinomial (quadratico, cubico, etc.)
  - O(2n) : exponential
  - O(n!) : factorial

Ordens de Crescimento

n	$\log_2 n$	n	$n \times \log_2 n$	n^2	n^3	2^n	n!
10	3.3	10	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9	?	?
10^4	13	10^4	1.3×10^4	10^8	10^{12}	?	?
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}	?	?
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}	?	?

Ordens mais comuns

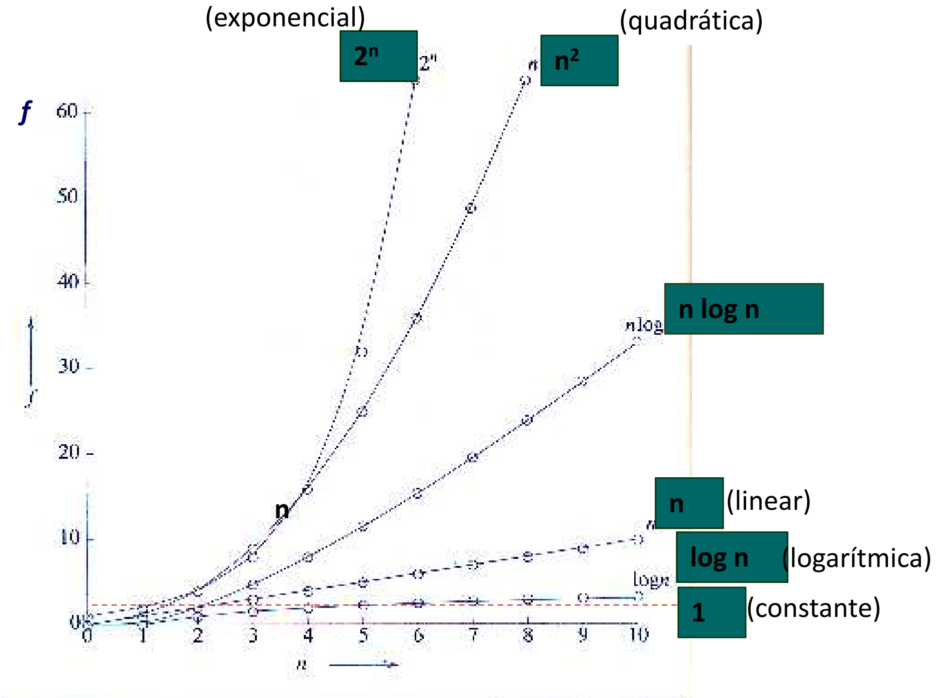


Figure 2.24 Plot of various functions

Fonte: Sahni, "Data Structures, Algorithms and Applications in C++"

Termo Dominante

- **Suponha que se usa N^3 para estimar $N^3+350N^2 + N$**
- **Para $N = 10000$**
  - Valor real = 1 003 500 010 000
  - Valor estimado = 1 000 000 000 000
  - Erro = 0.35% (não é significativo)
- **Para valores elevados de *N***
  - o termo dominante é indicativo do comportamento do algoritmo
- **Para valores pequenos de *N***
  - o termo dominante não é necessariamente indicativo do comportamento, mas geralmente, programas executam tão rapidamente que não importa

Eficiência da Pesquisa Sequencial

- **Eficiência temporal de SequentialSearch**
  - A operação realizada mais vezes é o teste da condição de continuação do ciclo **for**, no máximo **n+1** vezes (no caso de não encontrar **x**).
  - Se **x** existir no array, o teste é realizado aproximadamente **n/2** vezes em média (1 vez no melhor caso)
  - **T(n) = *O(n)*** (linear) no pior caso e no caso médio
- **Eficiência espacial de SequentialSearch**
  - Gasta o espaço das variáveis locais (incluindo argumentos)
  - Como os arrays são passados "por referência" (de facto o que é passado é o endereço do array), o espaço gasto pelas variáveis locais é constante e independente do tamanho do array
  - **S(n) = *O(1)*** (constante) em qualquer caso

Eficiência Temporal da Pesquisa Binária

- **Em cada iteração, o tamanho do sub-array a analisar é dividido por um factor de aproximadamente 2**
- **Ao fim de k iterações, o tamanho do sub-array a analisar é aproximadamente $n / 2^k$**
- **Se não existir no array o valor procurado, o ciclo só termina quando $n / 2^k \approx 1 \Leftrightarrow \log_2 n - k \approx 0 \Leftrightarrow k \approx \log_2 n$**
- **Assim, no pior caso, o nº de iterações é aproximadamente $\log_2 n$**

  $\Rightarrow$ **T(n) = *O(log n)* (logarítmico)**
- **É muito mais eficiente que a pesquisa sequencial, mas só é aplicável a arrays ordenados!**

Eficiência da Ordenação por Inserção

- **InsertSorted(v, n, x) :**
  - o nº de iterações do ciclo `for` é:
    - **1**, no melhor caso
    - **n**, no pior caso
    - **n/2**, em média
- **InsertionSort(v, n) :**
  - faz `InsertSorted(,1,)`, `InsertSorted(,2,)`, …, `InsertSorted(,n-1,)`
  - o nº total de iterações do ciclo `for` de `InsertSorted` é:
    - no **melhor caso**, 1 + 1 + ... + 1 (n-1 vezes) = n-1 ≈ **n**
    - no **pior caso**, 1 + 2 + ... + n-1 = (n-1)(1 + n-1)/2 = n(n-1)/2 ≈ **$n^2/2$**
    - em **média**, metade do anterior, isto é, aproximadamente **$n^2/4$**

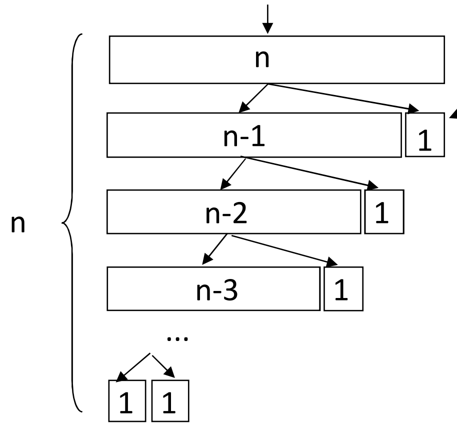
  ⇒ **T(n) = *O(n^2)*** (quadrático) (pior caso e média)

Eficiência da Ordenação por Partição

- **As operações realizadas mais vezes no passo de partição são as comparações efectuadas nos passos 2.3.1 e 2.3.2.**
- **No conjunto dos dois passos, o número de comparações efectuadas é:**
  - no mínimo **n** (porque todas as posições do array são analisadas)
  - no máximo **n+2** (correspondente à situação em que *i=j+1* no fim do passo 2.3.2)
- **Por conseguinte, o tempo de execução do passo de partição é *O(n)***
- **Para obter o tempo de execução do algoritmo completo, é necessário somar os tempos de execução do passo de partição, para o array inicial e para todos os sub-arrays aos quais o algoritmo é aplicado recursivamente**

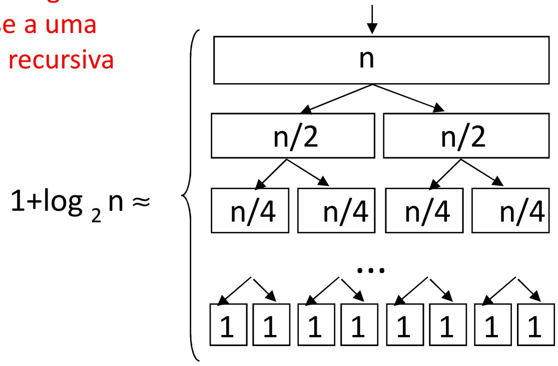
Eficiência da Ordenação por Partição (cont.)

Pior caso:



cada rectângulo refere-se a uma chamada recursiva

Melhor caso:



- **profundidade de recursão: n**
- **tempo de execução total (somando totais de linhas):**

  T(n) = O[n+n+(n-1) + ... +2]

  = O[n+(n-1)(n + 2)/2] = *O(n²)*

- profundidade de recursão: ≈ $1+\log_2 n$ (sem contar com a possibilidade de um elemento ser excluído dos sub-arrays esquerdo e direito)
- tempo de execução total (uma vez que a soma de cada linha é *n*):

  T(n) = $O[(1+\log_2 n)\ n]$ = ***O(n log n)***

Eficiência da Ordenação por Partição (cont.)

- **Prova-se que no caso médio (na hipótese de os valores estarem aleatoriamente distribuídos pelo array), o tempo de execução é da mesma ordem que no melhor caso, isto é:**

  T(n) = *O(n log n)*

- **O critério seguido para a escolha do *pivot* destina-se a tratar eficientemente os casos em que o array está inicialmente ordenado**

Comparação de tempos médios de execução (observados) de diversos algoritmos de ordenação

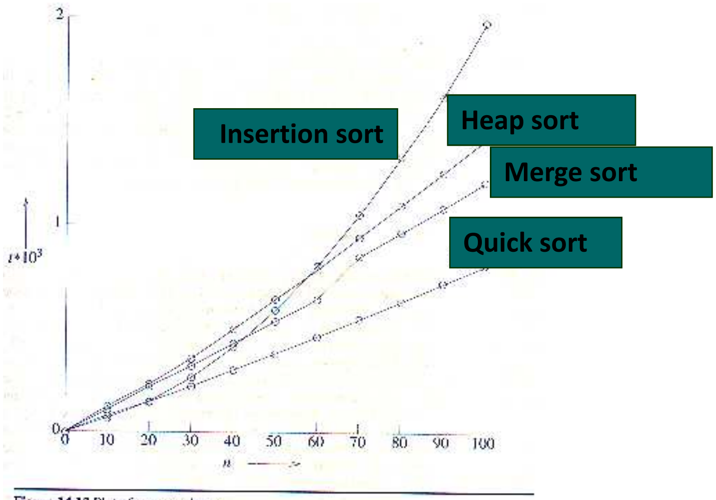


Figure 14.12 Plot of average times

Cada ponto corresponde à ordenação de 100 arrays de inteiros gerados aleatoriamente

Fonte: Sahni, "Data Structures, Algorithms and Applications in C++"

Método de ordenação por partição (quick sort) é na prática o mais eficiente, exceto para arrays pequenos (até cerca 20 elementos), em que o método de ordenação por inserção (insertion sort) é melhor!

Complexidade Espacial de QuickSort

- **O espaço de memória exigido por cada chamada de QuickSort, sem contar com chamadas recursivas, é independente do tamanho (*n*) do array**
- **O espaço de memória total exigido pela chamada de QuickSort, incluindo as chamadas recursivas, é pois proporcional à profundidade de recursão**
- **Assim, a <u>complexidade espacial</u> de QuickSort é:**
  - ***O(log n)*** no melhor caso (e no caso médio)
  - ***O(n)*** no pior caso
- **Em contrapartida, a <u>complexidade espacial</u> de InsertionSort é *O(1)***

Estudo de um caso: subsequência máxima

- **Problema:**
  - Dado um conjunto de valores (positivos e/ou negativos) $A_0, A_1, A_2, \ldots, A_{n-1}$, determinar a subsequência de maior soma
- **A subsequência de maior soma é zero se todos os valores são negativos**

- **Exemplos:**

  -2, <u>11, -4, 13</u>, -4, 2 -> Indíces 1 a 3, Valor = 20

  1, -3, <u>4, -2, -1, 6</u> -> Indíces 2 a 5, Valor = 7

$$\sum_{k=i}^{j} A_k$$

Subsequência máxima - cúbico

```
// MaxSubSum1: Calcula a Subsequência máxima utilizando três ciclos

template <class T>
T maxSubSum1(const vector<T> &vec)
{
    T maxSum = 0;
    for (int i = 0; i < vec.size(); i++)
       for (int j = i; j < vec.size(); j++)
       {
               T thisSum = 0;
               for (int k = i; k <= j; k++)
                      thisSum += vec[k];
               if (thisSum > maxSum) maxSum = thisSum;
       }
    return maxSum;
}
```

Subsequência máxima - cúbico

- **Análise**
  - ciclo de *N* iterações no interior de um outro ciclo de *N* iterações no interior de um outro ciclo de *N* iterações ⇒ ***O(N³)*** , algoritmo cúbico!
  - Valor estimado por excesso (factor de 6) pois alguns ciclos possuem menos de *N* iterações
- **Como melhorar**
  - Remover um ciclo
  - Ciclo mais interior não é necessário
  - *thisSum* para próximo *j* pode ser calculado facilmente a partir do antigo valor de *thisSum*

Subsequência máxima - quadrático

```
// MaxSubSum2: Calcula a Subsequência máxima utilizando dois ciclos

template <class T>
T maxSubSum2(const vector<T> &vec)
{
    T maxSum = 0;
    for (int i = 0 ; i < vec.size(); i++)
    {
        T thisSum = 0;
        for (int j = i; j < vec.size(); j++)
        {
            thisSum += vec[j];
            if (thisSum > maxSum) maxSum = thisSum;
        }
    }
    return maxSum;
}
```

Subsequência máxima - quadrático

- **Análise**
  - ciclo de *N* iterações no interior de um outro ciclo de *N* iterações $\Rightarrow$ ***O(N²)*** , algoritmo quadrático
- **É possivel melhorar?**
  - Algoritmo linear é melhor : tempo de execução é proporcional a tamanho de entrada (difícil fazer melhor)
    - Se A_{ij} é uma subsequência com custo negativo, A_{iq} com q>j não é a subsequência máxima

Subsequência máxima - linear

```
// MaxSubSum3: Calcula a Subsequência máxima utilizando um ciclo

template <class T>
T maxSubSum3(const vector<T> &vec)
{
    T thisSum = 0, maxSum = 0;
    for (int j=0; j < vec.size(); j++)
    {
       thisSum += vec[j];
       if (thisSum > maxSum)
            maxSum = thisSum;
       else if (thisSum < 0)
          thisSum = 0;
    return maxSum;
}
```

Subsequência máxima - recursivo

- **Método "divisão e conquista"**
  - Divide a sequência ao meio
  - A subsequência máxima está:
    a) na primeira metade
    b) na segunda metade
    c) começa na 1ª metade, vai até ao último elemento da 1ª metade, continua no primeiro elemento da 2ª metade, e termina num elemento da 2ª metade.
  - Calcula as três hipóteses e determina o máximo
  - a) e b) calculados recursivamente
  - c) realizado em dois ciclos:
    - percorrer a 1ª metade da direita para a esquerda, começando no último elemento
    - percorrer a 2ª metade da esquerda para a direita, começando no primeiro elemento

Subsequência máxima - recursivo

```
// MaxSubSumRec: Calcula a Subsequência máxima utilizando recursão

template <class T>
T maxSubSumRec(const vector<T> &vec,
        int left, int right)
{
  T maxLeftBorderSum = 0, maxRightBorderSum = 0;
  T leftBorderSum = 0, rightBorderSum = 0;
  int center = (left + right )/2;
  if (left == right)
        return (vec[left]>0 ? vec[left] : 0 );
  T maxLeftSum = maxSubSumRec(vec, left, center);
  T maxRightSum = maxSubSumRec(vec, center+1, right);
  …
```

Subsequência máxima - recursivo

```
   …
   for (int i = center ; i >= left ; i--)
   {
      leftBorderSum += vec[i];
      if (leftBorderSum > maxLeftBorderSum)
               maxLeftBorderSum = leftBorderSum;
   }

   for (int j = center+1 ; j <= right ; j++)
   {
      rightBorderSum += vec[j];
      if (rightBorderSum > maxRightBorderSum)
               maxRightBorderSum = rightBorderSum;
   }

   return max3(maxleftSum, maxRightSum,
               maxLeftBorderSum + maxRightBorderSum);
}
```


Subsequência máxima - recursivo

- **Análise**
  - Seja *T(N)* = tempo execução para problema tamanho N
  - *T(1)* = 1 (recorda-se que constantes não interessam)
  - *T(N)* = 2* *T(N/2)* + *N*
    - duas chamadas recursivas, cada uma de tamanho *N/2*. O tempo de execução de cada chamada recursiva é *T(N/2)*
    - tempo de execução de caso c) é *N*

Escola de Engenharia
Universidade do Minho

Subsequência máxima - recursivo

- **Análise:**

$$\begin{cases} T(N) = 2 * T(N/2) + N \\ T(1) = 1 \end{cases}$$

T(N/2) = 2* T(N/4) + N/2

T(N/4) = 2* T(N/8) + N/4

...

T(N) = 2*2*T(N/4) + 2*N/2 + N

T(N) = 2*2*2*T(N/8) + 2*2*N/4 +2*N/2 + N

$T(N) = 2^k * T(N/2^k) + kN$

$T(1) = 1 : \; N/2^k = 1 \Rightarrow k = \log_2 N$

$T(N) = N*1 + N* \log_2 N = O(N*logN)$

Escola de Engenharia
Universidade do Minho

Subsequência máxima - Comparação

- **Comparação para Valores Pequenos de N:**

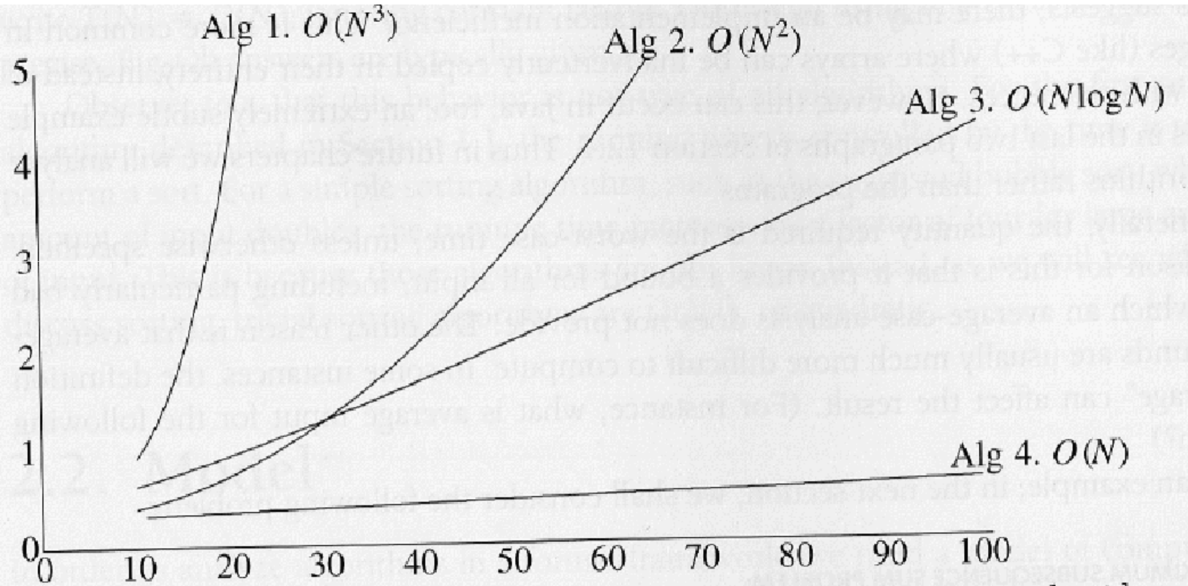


Figure 2.3 Plot (N vs. milliseconds) of various maximum subsequence sum algorithms

Fonte Weiss, 2010

Subsequência máxima - Comparação

- **Comparação para Valores Grandes de N:**

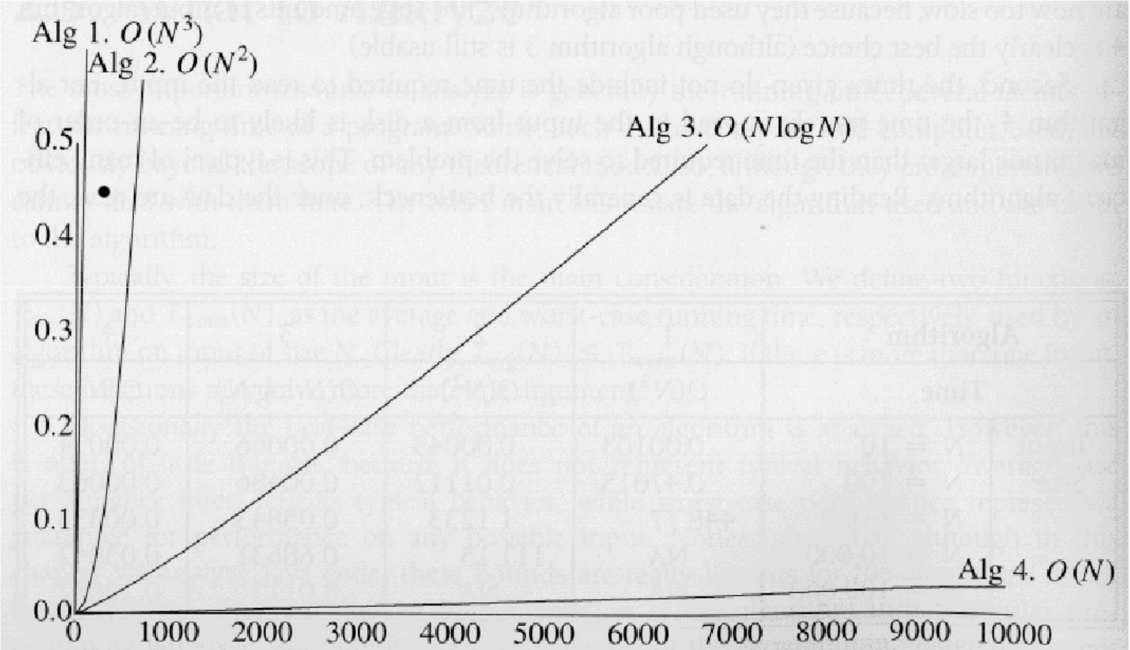


Figure 2.4 Plot (N vs. seconds) of various maximum subsequence sum algorithms

Fonte Weiss, 2010

Eficiência das Estruturas de Dados

	Acesso	Comentário
Pilha	Apenas ao elemento mais recente O(1)	Muito rápido
Fila	Apenas ao elemento menos recente O(1)	Muito rápido
Lista Ligada	Qualquer item O(N)	
Árvore de Pesquisa	Qualquer item por nome ou ordem O(log N)	Caso médio; em árvores especiais é pior caso
Tabela de Dispersão	Qualquer item por nome O(1)	Quase garantido
Fila de Prioridade	Acesso ao mínimo: O(1) Apagar mínimo: O(log N)	Inserção: O(1) caso médio, O(log N) pior caso

Complementos de Programação de Computadores – Aula 9
Análise de Complexidade de Algoritmos

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Reis, Rocha e Faria, 2007)