"Analysis of Algorithms"
de J.J. McConnell

Índice

Capítulo 1: Analysis Basics

1.1 WHAT ANALYSIS?

Este capítulo trata da análise de algoritmos e estruturas de dados, podendo ser realizada em termos de tempo (quantidade de operações executadas) ou espaço (quantidade de memória necessária).

1.2 WHAT TO COUNT AND CONSIDER

A decisão do que contar em uma análise algorítmica envolve duas etapas: escolher as operações significativas e distinguir entre aqueles essenciais para o algoritmo e as que são excessivas ou de manutenção.

Existem duas classes de operações geralmente escolhidas como operações significativas: comparação e aritmética. Operadores de comparação, como igual, não igual, menor que, maior que, menor ou igual e maior ou igual, são essenciais em algoritmos de busca e ordenação.

As operações aritméticas são contadas em dois grupos: aditivas (adição, subtração, incremento e decremento) e multiplicativas (multiplicação, divisão e módulo). Multiplicações são consideradas mais demoradas que adições. Além disso, outras operações, como logaritmos e funções geométricas, podem ser mais demoradas.

1.2.1 Casos a Considerar:

A escolha do input ao analisar um algoritmo pode impactar seu desempenho. Não se analisa apenas um conjunto de inputs; procura-se aqueles que permitem ao algoritmo ser mais rápido e mais lento, considerando também o desempenho médio.

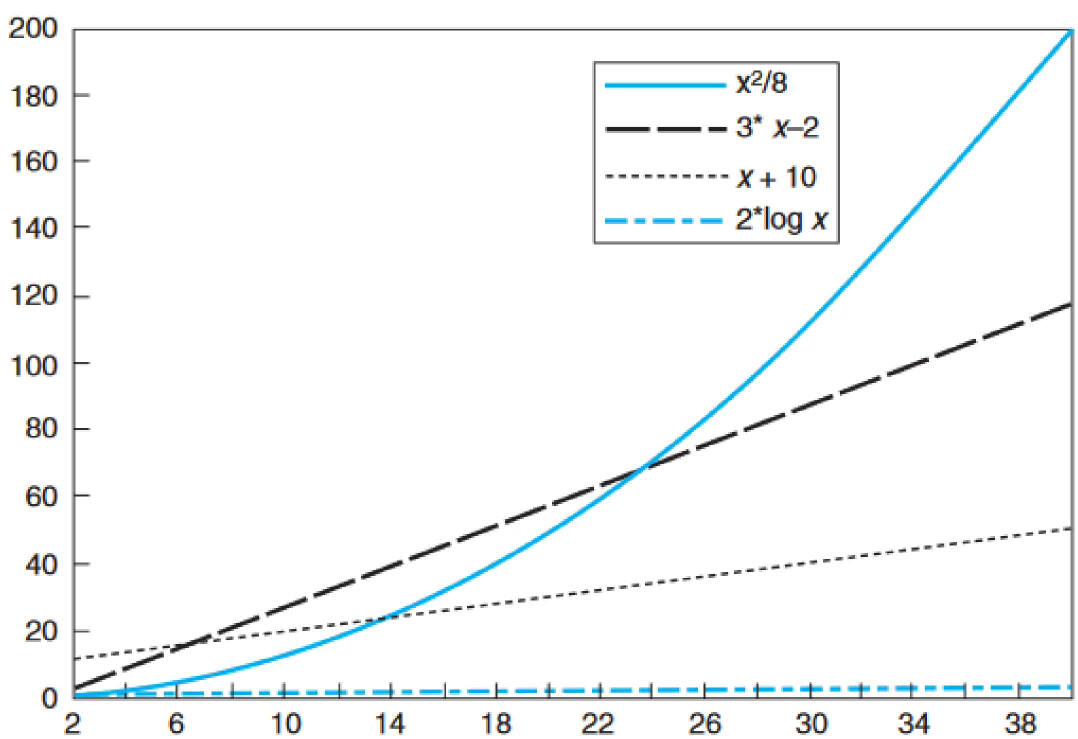
- **Melhor Caso:**
  - Entrada que exige o menor tempo possível.
  - Exemplo: Em um algoritmo de busca, a chave é encontrada na primeira comparação.
- **Pior Caso:**
  - Entrada que requer o maior esforço possível.
  - Exemplo: Em um algoritmo de busca, a chave está na última posição ou não está na lista.

- **Caso Médio:**
    - Análise complexa envolvendo diferentes grupos de inputs e suas probabilidades.

1.4 RATES OF GROWTH

Na análise de algoritmos, o número exato de operações não é crucial. Em vez disso, foca-se na taxa de aumento das operações à medida que o tamanho do problema aumenta, conhecida como a taxa de crescimento do algoritmo. O interesse reside nas tendências gerais quando os conjuntos de dados são grandes.

Ao analisar algoritmos, concentra-se na taxa geral de crescimento, não nos detalhes. Funções baseadas em x^2 crescem lentamente inicialmente, mas rapidamente à medida que o tamanho do problema aumenta. Funções baseadas em x crescem a uma taxa constante. Funções baseadas em log x crescem muito lentamente.



Na análise de algoritmos, interessa mais a classe de crescimento do algoritmo do que o número exato de operações. As classes comuns incluem funções quadráticas, lineares, logarítmicas, etc.

Classificação de Crescimento:

1. **Big Omega (Ω):**
   - Representa funções que crescem pelo menos tão rápido quanto outra função (limite inferior).
   - Exemplo: $\Omega(n^2)$ inclui funções que crescem pelo menos tão rápido quanto n^2.
2. **Big Oh (O):**
   - Representa funções que não crescem mais rápido que outra função (limite superior).
   - Exemplo: $O(n^2)$ inclui funções que crescem no máximo tão rápido quanto n^2.
3. **Big Theta (θ):**
   - Representa funções que crescem ao mesmo ritmo que outra função.
   - É a sobreposição de Big Omega e Big Oh.

Encontrar Big Oh:

- A função $g(n)$ pertence a $O(f(n))$ se o limite de $g(n) / f(n)$ for um número real menor que um valor constante ———————————— para n suficientemente grande.

Nota: Referencia para + a frente, "$g = O(f)$" é equivalente a "$g \in O(f)$".

Best case, Worst case, Average Case

$$B(n) = \min_{I \in D_n} t(I) \qquad W(n) = \max_{I \in D_n} t(I)$$

$$A(n) = \sum_{I \in D_n} p(I) \times t(I)$$

Alguns resultados úteis

- $\sum_{k=1}^{n} 1 = n$
- $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{k=1}^{n} \frac{1}{k} = \log n + \underbrace{\gamma}_{\substack{\text{Euler's constant} \\ \approx 0.577216}} + \frac{1}{2n} + O(n^{-2})$
- $\sum_{k=n}^{m} f(k) = \sum_{k=1}^{m} f(k) - \sum_{k=1}^{n-1} f(k)$

1.5 DIVIDE AND CONQUER ALGORITHMS

```
DivideAndConquer( data, N, solution )
data     a set of input values
N        the number of values in the set
solution the solution to this problem

if (N ≤ SizeLimit) then
   DirectSolution( data, N, solution )
else
   DivideInput( data, N, smallerSets, smallerSizes, numberSmaller )
   for i = 1 to numberSmaller do
      DivideAndConquer(smallerSets[i], smallerSizes[i], smallSolution[i])
   end for
   CombineSolutions(smallSolution, numberSmaller, solution)
end if
```

Este é um exemplo de um algoritmo de divisão e conquista. Os algoritmos de divisão e conquista resolvem problemas dividindo-os em subproblemas menores que são semelhantes ao problema original, resolvendo os subproblemas e combinando as soluções dos subproblemas para resolver o problema original.

Aqui está uma explicação passo a passo do código:

1. `if (N ≤ SizeLimit) then`: Verifica se o tamanho do problema é pequeno o suficiente para ser resolvido diretamente.
2. `DirectSolution(data, N, solution)`: Se o tamanho do problema for pequeno o suficiente, resolve o problema diretamente.
3. `DivideInput(data, N, smallerSets, smallerSizes, numberSmaller)`: Se o tamanho do problema for grande, divide o problema em subproblemas menores.
4. `for i = 1 to numberSmaller do`: Este é um loop que percorre cada subproblema.
5. `DivideAndConquer(smallerSets[i], smallerSizes[i], smallSolution[i])`: Resolve cada subproblema recursivamente usando o mesmo algoritmo de divisão e conquista.
6. `end for`: Este é o fim do loop.
7. `CombineSolutions(smallSolution, numberSmaller, solution)`: Combina as soluções dos subproblemas para formar a solução do problema original.
8. `end if`: Este é o fim da instrução condicional.

1. **Exemplo: Algoritmo Fatorial Recursivo:**

```
Factorial( N )
N          is the number we want the factorial for
Factorial returns an integer

If (N = 1) then
   return 1
else
   smaller = N - 1
   answer = Factorial( smaller )
   return (N * answer)
end if
```

- **Descrição:**
  - Resolve problemas dividindo-os em subproblemas menores.
  - Utiliza chamadas recursivas para tratar subconjuntos de dados.
  - Combina as soluções dos subproblemas para formar a solução final.

Eficiência Recursiva do Algoritmo:

- A eficiência é determinada pelos casos diretos, divisão do input, chamadas recursivas e combinação das soluções.

$$\mathrm{DAC}(N) = \begin{cases} \mathrm{DIR}(N) & \text{for } N \leq \text{SizeLimit} \\ \mathrm{DIV}(N) + \displaystyle\sum_{i=1}^{\text{numberSmaller}} \mathrm{DAC}(\text{smallerSizes}[i]) + \mathrm{COM}(N) & \text{for } N > \text{SizeLimit} \end{cases}$$

where DAC is the complexity of DivideAndConquer
DIR is the complexity of DirectSolution
DIV is the complexity of DivideInput
COM is the complexity of CombineSolutions

$$\mathrm{DAC}(N) = \begin{cases} N^2 & \text{for } N \le \text{SizeLimit} \\ \lg N + \sum_{i=1}^{8} \mathrm{DAC}(N/4) + N & \text{for } N > \text{SizeLimit} \end{cases}$$

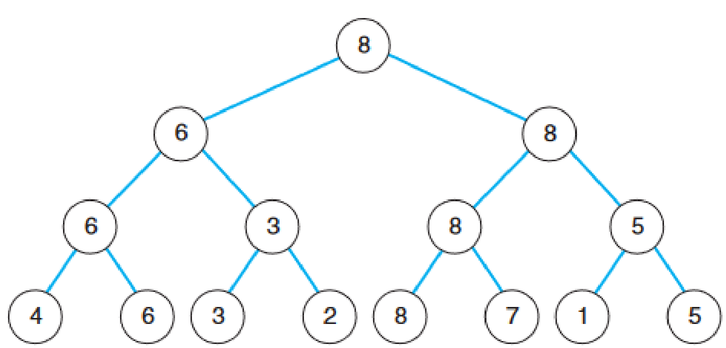
- Análise do exemplo do fatorial em relação à relação de recorrência.

$$\mathrm{Calc}(N) = \begin{cases} 0 & \text{for } N = 1 \\ 1 + \mathrm{Calc}(N-1) + 1 & \text{for } N > 1 \end{cases}$$

1.5.1 Método do Torneio:

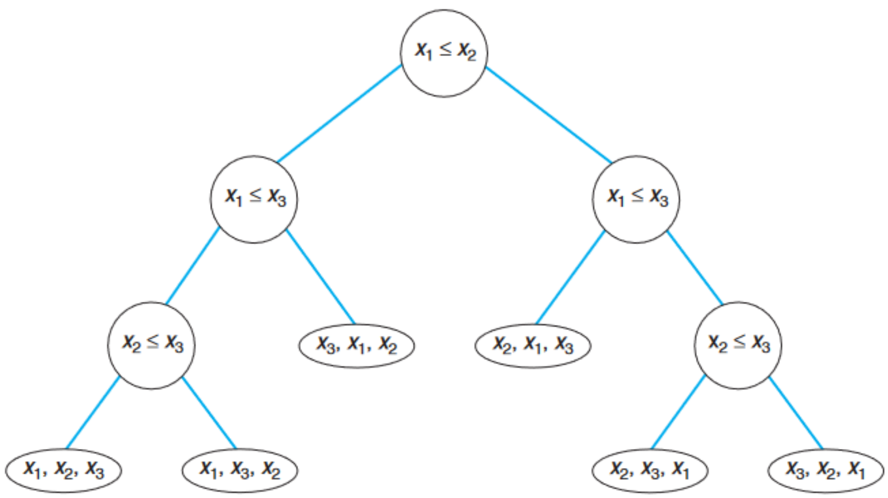
- **Construção da Árvore de Decisão:**
  - Cada comparação na árvore resulta em um "vencedor" e um "perdedor".
  - A busca descendente identifica os elementos que perderam para o maior.
- **Eficiência do Método do Torneio:**
  - Destaca a estrutura equilibrada da árvore.
  - Explica como encontrar o segundo maior elemento usando o método.

Cada comparação produz um “vencedor” e um “perdedor”. Os perdedores são eliminados e apenas os vencedores sobem na árvore. Cada elemento, exceto o maior, deve “perder” uma comparação. Portanto, construir a árvore do torneio levará N - 1 comparações. O segundo maior elemento só poderia ter perdido para o maior elemento. Descemos a árvore e obtemos o conjunto de elementos que perderam para o maior.



1.5.2 Limites Inferiores: Árvores de Decisão e Complexidade Ótima

- **Árvores de Decisão em Algoritmos de Ordenação:**
  - Uso de árvores de decisão para analisar algoritmos de ordenação.
  - Cada nó interno representa uma comparação entre dois elementos.
  - o caminho mais longo representa o pior caso. O melhor caso é o caminho mais curto. O caso médio é o número total de arestas na árvore de decisão dividido pelo número de folhas na árvore.



Podemos novamente usar uma árvore binária para nos ajudar a analisar o processo de ordenação de uma lista de três números. Podemos construir uma árvore binária para o processo de ordenação rotulando cada nó interno com os dois elementos da lista que seriam comparados.

- **Análise de Limites Inferiores para Algoritmos de Ordenação:**
    - Destaca a necessidade de determinar o número mínimo absoluto de operações para resolver um problema.
    - Utilização de uma árvore de decisão para analisar a ordenação de uma lista de três elementos.
    - Expressa a complexidade mínima como O(N lg N), onde N é o número de elementos na lista.

Capítulo 2: Searching and Selection Algorithms

2.1 Sequential Search

- **Objetivo:** Encontrar um elemento específico (alvo) em uma lista.
- **Abordagem:** Percorre a lista elemento por elemento até encontrar uma correspondência.
- **Caso Padrão:** Lista não ordenada, pois existem algoritmos mais eficientes para listas ordenadas.
- **Retorno:** Índice do elemento encontrado; 0 se o alvo não estiver na lista.

```
SequentialSearch( list, target, N )
list      the elements to be searched
target    the value being searched for
N         the number of elements in the list

for i = 1 to N do
   if (target = list[i])
      return i
   end if
end for
return 0
```

Este é um exemplo de um algoritmo de pesquisa binária. A pesquisa binária é um algoritmo de pesquisa eficiente que procura um item específico em uma lista ordenada. Funciona dividindo a lista pela metade e comparando o item do meio com o valor de destino. Se o valor do meio for igual ao valor de destino, o algoritmo retorna o índice do meio. Se o valor do meio for menor que o valor de destino, o algoritmo repete a pesquisa na metade superior da lista. Se o valor do meio for maior que o valor de destino, o algoritmo repete a pesquisa na metade inferior da lista. Este processo continua até que o valor de destino seja encontrado ou a sublista se torne vazia.

Aqui está uma explicação passo a passo do código:

1. `start = 1` e `end = N`: Inicializa as variáveis de início e fim para os extremos da lista.
2. `while start <= end do`: Continua a pesquisa enquanto a sublista não estiver vazia.
3. `middle = (start + end) / 2`: Calcula o índice do meio.
4. `select (Compare (list [middle) , target)) from`: Compara o valor do meio com o valor de destino.
5. `case -1:start = middle + 1`: Se o valor do meio for menor que o valor de destino, move o início para depois do meio.
6. `case 0: return middle`: Se o valor do meio for igual ao valor de destino, retorna o índice do meio.
7. `case 1: end = middle - 1`: Se o valor do meio for maior que o valor de destino, move o fim para antes do meio.
8. `return 0`: Se a pesquisa não encontrar o valor de destino, retorna 0.

2.1.1 Análise do Pior Caso

- **Cenários de Pior Caso:**
    1. O alvo é o último elemento na lista.
    2. O alvo não está na lista.
- **Número de Comparações:** Sempre leva N comparações, onde N é o número de elementos na lista.

2.1.2 Análise do Caso Médio

- **Caso Bem-Sucedido:**
    - Probabilidade igual de o alvo estar em qualquer posição da lista.
    - Número médio de comparações é (N + 1) / 2.

$$A(N) = \frac{1}{N}\sum_{i=1}^{N} i$$

$$A(N) = \frac{1}{N} * \frac{N(N+1)}{2}$$

$$A(N) = \frac{N+1}{2}$$

Foto do livro

- **Incluindo Possibilidade de Falha:**
  - Número médio de comparações é (N + 1) / 2 + 1/2.

$$A(N) = \left(\frac{1}{N+1}\right) * \left[\left(\sum_{i=1}^{N} i\right) + N\right]$$

$$A(N) = \left(\frac{1}{N+1}\sum_{i=1}^{N} i\right) + \left(\frac{1}{N+1} * N\right)$$

$$A(N) = \left(\frac{1}{N+1} * \frac{N(N+1)}{2}\right) + \frac{N}{N+1}$$

$$A(N) = \frac{N}{2} + \frac{N}{N+1} = \frac{N}{2} + 1 - \frac{1}{N+1}$$

$$A(N) \approx \frac{N+2}{2} \qquad \text{(As } N \text{ gets very large, } \frac{1}{N+1} \text{ becomes almost 0.)}$$

Foto do livro

2.2 Binary Search

- **Objetivo:** Buscar em uma lista ordenada.
- **Abordagem:** Compara o alvo com o elemento no meio e elimina metade da lista a cada iteração.
- **Loop:** Continua até encontrar o alvo ou reduzir a lista a zero.
- **Retorno:** Índice do elemento encontrado; 0 se o alvo não estiver na lista.

```
BinarySearch( list, target, N )
list      the elements to be searched
target    the value being searched for
N         the number of elements in the list

start = 1
end = N
while start ≤ end do
   middle = (start + end) / 2
   select (Compare(list[middle], target)) from
      case -1:start = middle + 1
      case  0:return middle
      case  1:end = middle - 1
   end select
end while
return 0
```

Este é um exemplo de um algoritmo de pesquisa binária. A pesquisa binária é um algoritmo de pesquisa eficiente que procura um item específico em uma lista ordenada. Funciona dividindo a lista pela metade e comparando o item do meio com o valor de destino. Se o valor do meio for igual ao valor de destino, o algoritmo retorna o índice do meio. Se o valor do meio for menor que o valor de destino, o algoritmo repete a pesquisa na metade superior da lista. Se o valor do meio for maior que o valor de destino, o algoritmo repete a pesquisa na metade inferior da lista. Este processo continua até que o valor de destino seja encontrado ou a sublista se torne vazia.

Aqui está uma explicação passo a passo do código:

1. `start = 1` e `end = N`: Inicializa as variáveis de início e fim para os extremos da lista.
2. `while start <= end do`: Continua a pesquisa enquanto a sublista não estiver vazia.
3. `middle = (start + end) / 2`: Calcula o índice do meio.
4. `select (Compare (list [middle) , target)) from`: Compara o valor do meio com o valor de destino.
5. `case -1:start = middle + 1`: Se o valor do meio for menor que o valor de destino, move o início para depois do meio.
6. `case 0: return middle`: Se o valor do meio for igual ao valor de destino, retorna o índice do meio.
7. `case 1: end = middle - 1`: Se o valor do meio for maior que o valor de destino, move o fim para antes do meio.
8. `return 0`: Se a pesquisa não encontrar o valor de destino, retorna 0.

2.2.1 Árvore de Decisão

- **Decisão Estruturada:** Construir uma árvore de decisão para o processo de busca.
- **Nós da Árvore:** Elementos verificados em cada passo.
- **Subárvores:** Elementos verificados quando o alvo é menor ou maior que o elemento atual.
- **Equilíbrio da Árvore:** A escolha do meio da lista garante uma árvore relativamente equilibrada.
- **Número de Comparativos:** No máximo, lg(N + 1) comparações, onde N é o número de elementos na lista.

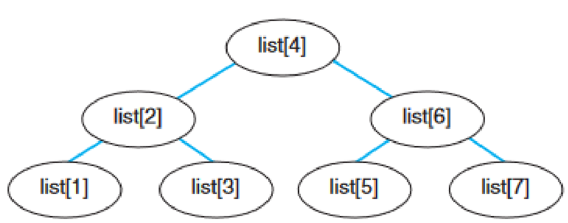


Foto do livro

2.2.2 Análise do Caso Médio

- **Situações Analisadas:**
  1. O alvo está sempre na lista.
  2. O alvo pode não estar na lista.
- **Probabilidades:** Caso 1 - Cada localização tem probabilidade 1/N.
- **Número Médio de Comparativos:**
  - A análise é baseada na estrutura da árvore de decisão.
  - Equação para o caso em que o alvo está na lista:

$$A(N) = \frac{k(N+1)}{N} - 1$$

$$A(N) = \frac{k * N + k}{N} - 1$$

As N gets larger, k/N becomes zero, giving

$$A(N) \approx k - 1 \qquad \text{for } N = 2^k - 1$$

$$A(N) \approx \lg(N+1) - 1 \qquad \text{for } N = 2^k - 1$$

Foto do livro

  - Equação para o caso em que o alvo não está na lista:

$$A(N) \approx k - \frac{1}{2} = \lg(N+1) - \frac{1}{2} \quad \text{for } N = 2^k - 1$$

Foto do livro

2.3 Seleção

- **Objetivo:**
  - Exploração de situações em que a busca é baseada em propriedades específicas, não apenas em valores.
  - Foco na identificação do maior, menor ou valor mediano, especialmente na busca pelo KK-ésimo maior valor em uma lista.

Algoritmo FindKthLargest

```
FindKthLargest( list, N, K )
list   the values to look through
N      the size of the list
K      the element to select

for i = 1 to K do
   largest = list[1]
   largestLocation = 1

   for j = 2 to N-(i-1) do
      if list[j] > largest then
         largest = list[j]
         largestLocation = j
      end if
   end for
   Swap( list[N-(i-1)], list[largestLocation] )
end for
return largest
```

Este é um exemplo de um algoritmo para encontrar o K-ésimo maior elemento em uma lista. O algoritmo funciona selecionando o maior elemento da lista K vezes. Aqui está uma explicação passo a passo do código:

1. `for i = 1 to K do`: Este é um loop que será executado K vezes. Cada iteração do loop seleciona o próximo maior elemento da lista.
2. `largest = list[1]` e `largestLocation = 1`: Inicializa o maior elemento e sua localização para o primeiro elemento da lista.
3. `for j = 2 to N-(i-1) do`: Este é um loop interno que percorre cada elemento restante da lista.
4. `if list[j] > largest then`: Verifica se o elemento atual é maior que o maior elemento encontrado até agora.
5. `largest = list[j]` e `largestLocation = j`: Se o elemento atual for maior, atualiza o maior elemento e sua localização.
6. `Swap(list[N-(i-1)], list[largestLocation])`: Troca o maior elemento encontrado com o elemento na posição `N-(i-1)`. Isso move o maior elemento para o final da lista.
7. `end for`: Este é o fim do loop interno.
8. `return largest`: Após K iterações, o K-ésimo maior elemento será o último elemento na lista. O algoritmo retorna este elemento.

- **Descrição:**
  - Proposta de algoritmo para encontrar o K-ésimo maior valor em uma lista.
  - Estratégia inicial: ordenar a lista em ordem decrescente e selecionar o valor na posição K.
  - Alternativa mais eficiente: seleção iterativa, movendo os maiores valores para o final da lista.

- **Complexidade:**
  - Cada passo faz N−(i+1) comparações.

On the K^{th} pass, we do $N - K$ comparisons. So, to find the K^{th} largest element, we will do

$$\sum_{i=1}^{K} N - i \;=\; N * K - \frac{K * (K-1)}{2}$$

Foto do livro

- No total, O(K·N)O(K·N) comparações.
- Observação: Eficiente para K próximo às extremidades da lista; estratégia mais eficaz para K no meio:

Algoritmo KthLargestRecursive

```
KthLargestRecursive( list, start, end, K )
list  the list of values
start  the index of the first value to consider
end    the value of the last value to consider
K      the element of this list that we want.

if start < end then
   Partition( list, start, end, middle )
   if middle = K then
      return list[middle]
   else
      if K < middle then
         return KthLargestRecursive( list, middle+1, end, K )
      else
         return KthLargestRecursive( list, start, middle-1, K-middle )
      end if
   end if
end if
```

Este é um exemplo de um algoritmo recursivo para encontrar o K-ésimo maior elemento em uma lista. O algoritmo usa a técnica de partição (como no algoritmo de ordenação rápida) para dividir a lista em duas partes, de modo que todos os elementos à esquerda do elemento do meio sejam menores que ele e todos os elementos à direita do elemento do meio sejam maiores que ele.

Aqui está uma explicação passo a passo do código:

1. `if start < end then`: Verifica se a lista ou sublista ainda tem elementos para serem verificados.
2. `Partition(list, start, end, middle)`: Particiona a lista ou sublista em torno do elemento do meio.
3. `if middle = K then`: Verifica se o índice do elemento do meio é igual a K.
4. `return list[middle]`: Se o índice do elemento do meio for igual a K, retorna o elemento do meio.
5. `if K < middle then`: Se K for menor que o índice do elemento do meio, repete a pesquisa na sublista à direita do elemento do meio.
6. `return KthLargestRecursive(list, middle+1, end, K)`: Chama a função recursivamente com a sublista à direita do elemento do meio.
7. `return KthLargestRecursive(list, start, middle-1, K-middle)`: Se K for maior que o índice do elemento do meio, repete a pesquisa na sublista à esquerda do elemento do meio e ajusta K para levar em conta os elementos à direita do elemento do meio.
8. `end if`: Este é o fim da instrução condicional.

- **Descrição:**
  - Algoritmo recursivo mais eficiente para encontrar o K-ésimo maior valor.
  - Utiliza o conceito de partição para dividir a lista, reduzindo a complexidade.
  - Exploração da partição média: aproximadamente 2N comparações.

Observações Finais

- **Eficiência:**
  - Estratégias mais rápidas para K próximo ao início ou final da lista.
  - Algoritmo recursivo eficaz para K no meio, explorando a partição média.
- **Partição:**
  - Prévia menção sobre a exploração detalhada da partição no quicksort no Capítulo 3.

Capítulo 3: Sorting Algorithms

3.1 Ordenação por Inserção

- **Ideia Básica:**
  - Adição eficiente de um novo elemento a uma lista ordenada.
  - Evitar adicionar em qualquer lugar e, em seguida, reordenar a lista inteira.
  - Considera a primeira posição da lista como uma lista ordenada de tamanho 1.

Algoritmo InsertionSort

```
InsertionSort( list, N )
list   the elements to be put into order
N      the number of elements in the list

for i = 2 to N do
   newElement = list[ i ]
   location = i - 1
   while (location ≥ 1) and (list[ location ] > newElement) do
      // move any larger elements out of the way
      list[ location + 1] = list[ location ]
      location = location - 1
   end while
   list[ location + 1 ] = newElement
end for
```

Este é um exemplo de um algoritmo de ordenação por inserção. A ordenação por inserção é um algoritmo simples que constrói a lista final de elementos ordenados um item de cada vez. É muito parecido com a maneira como você ordena cartas de baralho em suas mãos.

Aqui está uma explicação passo a passo do código:

1. `for i = 2 to N do`: Este é um loop que percorre cada elemento da lista a partir do segundo elemento. `i` é o índice do elemento atual.
2. `newElement = list[i]`: Armazena o valor do elemento atual em uma variável `newElement`.
3. `location = i - 1`: Inicializa a variável `location` para o índice do elemento anterior.
4. `while (location ≥ 1) and (list[location] > newElement) do`: Este é um loop interno que continua enquanto `location` é maior ou igual a 1 e o elemento na posição `location` é maior que `newElement`.
5. `list[location + 1] = list[location]`: Move o elemento na posição `location` uma posição para a direita.
6. `location = location - 1`: Diminui `location` em 1.
7. `end while`: Este é o fim do loop interno.
8. `list[location + 1] = newElement`: Insere `newElement` na posição correta na lista ordenada.
9. `end for`: Este é o fim do loop externo.

Insertion Sort

```
void insertionSort( int a[], int n ) {
        for( int  i = 1; i < n; i++ )
                if( a[i] < a[i – 1] )
                        insertElement( a, i, a[i] );
}
```

Talvez mais fácil de entender que a anterior.

- **Descrição:**
  - Itera sobre os elementos, inserindo cada novo elemento na posição correta.
  - O processo envolve deslocar elementos maiores para abrir espaço.

- **Complexidade:**
  - Cada iteração compara o novo elemento a, no máximo, ii elementos anteriores.
  - Pior caso: O(N^2) comparações.
  - Pior caso ocorre quando a lista está em ordem decrescente inicialmente.

3.1.1 Análise do Pior Caso

- **Caso Pior:**
  - Cada novo elemento adicionado é menor que todos os elementos já ordenados.
  - O loop interno executa o máximo de trabalho quando o novo elemento é adicionado no início da lista.

- **Complexidade no Pior Caso:**
  - Soma das comparações: N+(N−1)+(N−2)+…+1N+(N−1)+(N−2)+…+1.

$$W(N) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2}$$

$$W(N) = O(N^2)$$

Foto do livro

3.1.2 Análise do Caso Médio

A análise do caso médio é realizada em duas etapas. Primeiro, determinamos o número médio de comparações necessário para posicionar um elemento. Em seguida, usamos esse resultado para calcular o número médio total de operações.

- **Número Médio de Comparações para o i-ésimo Elemento:**
  - Adicionando o i-ésimo elemento à parte ordenada, leva no máximo i comparações.
  - O pior caso ocorre quando o novo elemento é menor que todos os elementos já ordenados.
  - Em média, há i+1 posições possíveis para o i-ésimo elemento.

$$A_i = \frac{1}{i+1} \sum_{p=0}^{i} p = \frac{1}{i+1} \cdot \frac{i(i+1)}{2} = \frac{i}{2}$$

Foto do chat 1.1, talvez duvidoso. Há no livro, mas é mais confuso.

- **Número Médio Total de Operações:**
  - Somamos os resultados para todos os elementos de 1 a N−1.

$$A(N) = \sum_{i=1}^{N-1} A_i = \sum_{i=1}^{N-1} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{N-1} i = \frac{1}{2} \cdot \frac{(N-1)N}{2} = \frac{N(N-1)}{4}$$

Foto do chat1.2, talvez duvidoso. Há no livro, mas é mais confuso.

  - Isso é equivalente a O(N^2), mas a análise detalhada mostra uma constante menor.

- **Simplificação:**

$$A(N) = \frac{N(N-1)}{4} = \frac{N^2}{4} - \frac{N}{4} = O(N^2) - O(N)$$

Foto do chat1.3, talvez duvidoso. Há no livro, mas é mais confuso.

3.2: Bubble Sort

```
BubbleSort( list, N )
list  the elements to be put into order
N     the number of elements in the list

numberOfPairs = N
swappedElements = true
while swappedElements do
   numberOfPairs = numberOfPairs - 1
   swappedElements = false
   for i = 1 to numberOfPairs do
      if list[ i ] > list[ i + 1 ] then
         Swap( list[i], list[i + 1] )
         swappedElements = true
      end if
   end for
end while
```

Este é um exemplo de um algoritmo de ordenação de bolha. A ordenação de bolha é um algoritmo simples que compara repetidamente pares adjacentes de elementos e os troca se estiverem na ordem errada. O algoritmo continua até que não seja mais necessário trocar elementos, o que indica que a lista está ordenada.

Aqui está uma explicação passo a passo do código:

1. `numberOfPairs = N` e `swappedElements = true`: Inicializa o número de pares a serem comparados e uma variável booleana para rastrear se algum elemento foi trocado na última passagem pela lista.
2. `while swappedElements do`: Continua a ordenação enquanto elementos estão sendo trocados.
3. `numberOfPairs = numberOfPairs - 1`: Diminui o número de pares a serem comparados, pois o maior elemento será movido para a posição correta após cada passagem pela lista.
4. `swappedElements = false`: Reinicia a variável booleana para a próxima passagem pela lista.
5. `for i = 1 to numberOfPairs do`: Este é um loop que percorre cada par de elementos na lista.
6. `if list[i] > list[i + 1] then`: Compara um par de elementos adjacentes.
7. `Swap(list[i], list[i + 1])`: Se o elemento à esquerda for maior que o elemento à direita, troca os elementos.
8. `swappedElements = true`: Marca que uma troca ocorreu.
9. `end if` e `end for`: Estes são os fins da instrução condicional e do loop interno.
10. `end while`: Este é o fim do loop externo.

O algoritmo Bubble Sort faz várias passagens pela lista de elementos, comparando valores adjacentes e trocando-os se estiverem fora de ordem. A cada passagem, os valores mais altos se movem para o final da lista, enquanto os menores se movem para o início. O processo continua até que nenhum elemento seja trocado em uma passagem, indicando que a lista está ordenada.

3.2.1 Análise do Melhor Caso

O melhor caso ocorre quando a lista já está ordenada. Nesse caso, o algoritmo faz N−1N−1 comparações na primeira passagem. As comparações diminuem em cada passagem subsequente, e se nenhuma troca ocorrer em uma passagem, o algoritmo para. Assim, o melhor caso é N−1 comparações.

3.2.2 Análise do Pior Caso

O pior caso ocorre quando a lista está em ordem reversa. Cada passagem move o maior elemento para a posição correta, mas o próximo maior elemento é movido para a segunda posição, e assim por diante. Isso resulta em N(N−1)/ 2 comparações no pior caso.

$$W(N) = \sum_{i=N-1}^{1} i = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2}$$

$$W(N) \approx \frac{1}{2}N^2 = O(N^2)$$

Foto do livro

3.2.3 Análise do Caso Médio

Para o caso médio, consideramos que é igualmente provável que em cada passagem não haja trocas. O número médio de comparações é dado por uma série, resultando em O(N^2)), mas com uma constante menor do que o pior caso.

$$A(N) = \frac{1}{N-1}\sum_{i=1}^{N-1} C(i)$$

$$C(i) = \frac{(N-1)N}{2} - \frac{(i-1)i}{2} = \frac{N^2 - N - i^2 + i}{2}$$

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

$$A(N) = \frac{1}{N-1}\left[(N-1) * \frac{N^2 - N}{2} + \sum_{i=1}^{N-1}\left(\frac{-i^2 + i}{2}\right)\right]$$

$$A(N) = \frac{N^2 - N}{2} + \frac{1}{2(N-1)}\left[\sum_{i=1}^{N-1} -i^2 + \sum_{i=1}^{N-1} i\right]$$

$$A(N) = \frac{N^2 - N}{2} + \frac{1}{2(N-1)}\left[-\frac{(N-1)N(2N-1)}{6} + \frac{(N-1)N}{2}\right]$$

$$A(N) = \frac{N^2 - N}{2} - \frac{N(2N-1)}{12} + \frac{N}{4}$$

$$A(N) = \frac{6N^2 - 6N - 2N^2 + N + 3N}{12}$$

$$A(N) = \frac{4N^2 - 2N}{12}$$

$$A(N) \approx \frac{1}{3}N^2 = O(N^2)$$

Foto do livro que só um DEUS percebe.