Técnicas de Projeto (Parte 1)
Projeto e Análise de Algoritmo

Felipe Domingos da Cunha

Pontifícia Universidade Católica de Minas Gerais

Introdução

- O projeto de algoritmos requer abordagens adequadas:
  - A forma como um algoritmo aborda o problema pode levar a um desempenho ineficiente,
  - Em certo casos, o algoritmo pode não conseguir resolver o problema em tempo viável.

- Serão apresentados os principais paradigmas a serem seguidos durante o projeto de algoritmos, os quais levam a abordagens adequadas de projeto.

Técnicas de Projeto

1) Força Bruta

2) Transformar e Conquistar

3) Decrementar e Conquistar

Força Bruta

- É a mais simples das técnicas de projeto
- Solução direta, geralmente baseada no enunciado do problema
  - Pode ser recursiva, mas na maioria das vezes é iterativa.
- É fácil de aplicar e muitas vezes surge como idéia intuitiva e pouco elaborada
- Pode exigir grande esforço computacional, mas os algoritmos são fáceis de entender

Força Bruta

- Muitas vezes são uma primeira versão para soluções mais elaboradas.
- Aplicável a uma ampla variedade de problemas.
  - Exemplos: cálculo do fatorial de um número, busca sequencial, ordenação pelo método da bolha, multiplicação de matrizes.
- Útil para o desenvolvimento rápido de algoritmos que operem sobre uma entrada pequena ou que serão executados poucas vezes.

Força Bruta: BubbleSort

```
Para i:=1 até n -1
Para j:=1 até n - i
  Se v [ j +1 ] < v [ j ] então troque v [ j ] com v [ j+1]
```

- Análise do algoritmo em função do número de comparações:
- O algoritmo é ótimo?

Força Bruta: Casamento de Padrões em Strings

```
String s[1..n], padrão p[1..m],  m<n:

Para i:=1 até n – m +1
      j:=1
Enquanto j<=m e p [ j ] = s [ j + i -1]
      j:= j + 1
Se j > m retorne i senão retorne –1
```

- Análise do algoritmo em função do número de comparações:
- O algoritmo é ótimo?
- Existem algoritmos de força bruta que são ótimos?

Força Bruta: Busca Exaustiva

- Aplicado a problemas de otimização com número de soluções exponencial.
- Todas as possíveis soluções são geradas e a melhor é selecionada.
- Exemplos:
  - Caixeiro viajante;
  - Problema da mochila;
  - Preenchimento de containers, etc.

Transformar e Conquistar

- Esta técnica compreende dois estágios:

1) No estágio de transformação, a instância do problema é transformada para ser mais fácil encontrar uma solução.

2) No segundo estágio, a instância transformada é resolvida.

- A técnica pode ser usada para o projeto de algoritmos recursivos e não recursivos.

Transformar e Conquistar

- Existem 3 variações da técnica:

1) **Simplificação**: transformação para uma instância mais simples ou conveniente do mesmo problema.
   - Exemplo: pré-ordenação.
2) **Mudança de representação**: transformação para uma representação diferente, na qual o problema é mais facilmente resolvido.
   - Exemplos: heapsort, transformada rápida de Fourier.
3) **Redução:** transformação para uma instância de um problema diferente para o qual já existe um algoritmo eficaz.
   - Exemplo: problemas de grafos.

Transformar e Conquistar: Pré-ordenação

- Muitos problemas envolvendo listas são mais simples de serem resolvidos quando a lista já está ordenada.

- Exemplo: verificar se existem elementos repetidos em um arranjo.

Transformar e Conquistar: Pré-ordenação

- Muitos problemas envolvendo listas são mais simples de serem resolvidos quando a lista já está ordenada.

- Exemplo: verificar se existem elementos repetidos em um arranjo.
  - Por força bruta, o algoritmo é $\Theta(n^2)$ no pior caso ($O(n^2)$ no caso geral).

Transformar e Conquistar: Pré-ordenação

- Muitos problemas envolvendo listas são mais simples de serem resolvidos quando a lista já está ordenada.

- Exemplo: verificar se existem elementos repetidos em um arranjo.
  - Por força bruta, o algoritmo é $\Theta(n^2)$ no pior caso ($O(n^2)$ no caso geral).
  - Alternativa: ordenar o arranjo e verificar elementos adjacentes $\Theta(n \log n + n) = \Theta(n \log n)$ no pior caso.

Transformar e Conquistar: HeapSort

- Um *Heap* é uma estrutura de dados parcialmente ordenada que é adequada para a implementação de filas de prioridade.

- Fila de prioridade é um conjunto de itens com uma característica ordenável chamada prioridade, contendo as seguintes operações:
  - Encontrar um elemento com a prioridade mais alta
  - Remover um elemento com a prioridade mais alta
  - **Adicionar um novo item ao conjunto**

Transformar e Conquistar: HeapSort

- Definição: Um heap é uma árvore binária essencialmente completa com chaves atribuídas aos seus nós, onde a chave de um nó é maior ou igual a chave dos seus nós-filhos.

- Propriedades:
  a) A altura de um heap com n nós é $\lfloor \lg n \rfloor$
  b) A raiz do heap sempre contém o maior elemento
  c) Cada sub-árvore é também um heap

Transformar e Conquistar: HeapSort

- Um heap pode ser implementado eficientemente como um arranjo:
  - Os filhos de um nodo na posição i estão nas posições $2i$ e $2i+1$
  - O pai de um nodo na posição i está na posição $i/2$

Transformar e Conquistar: HeapSort

- O uso da estrutura heap permite que:
  - O elemento máximo do conjunto seja determinado e corretamente posicionado no vetor em tempo constante, trocando-se o primeiro elemento do heap com o último.
  - O trecho restante do vetor (do índice 1 ao $n-1$), que pode ter deixado de ter a estrutura de heap, volte a tê-la com número de trocas de elementos proporcional à altura da árvore.

Transformar e Conquistar: HeapSort

- O algoritmo Heapsort consiste da construção de um heap seguida de sucessivas trocas do primeiro com o último elemento e rearranjos do heap:

```
Heapsort(A)
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.
1. ConstroiHeap(A, n)
2. Para ultimo:=n até 2 faça
◁       t := A[ultimo]
  A[ultimo] := A[1]
  A[1] := t
◁     AjustaHeap(A, 1, ultimo)
```

Transformar e Conquistar: HeapSort

```
AjustaHeap(A, i , n)

Entrada: Vetor A de n números inteiros com estrutura
de heap, exceto, talvez, pela sub-árvore de raiz i
Saída: Vetor A com estrutura de heap

se 2i ≤ n e A[2i ] ≥ A[i ] então
   maximo := 2i
senão maximo := i
se 2i + 1 ≤ n e A[2i + 1] ≥ A[maximo] então
  maximo := 2i + 1
se maximo <> i então
  t := A[maximo]
  A[maximo] := A[i ]
  A[i ] := t
  AjustaHeap(A, maximo, n)
```

Transformar e Conquistar: HeapSort

- Análise: (em função da altura da árvore, h)
  - Quantas comparações e quantas trocas são executadas no pior caso na etapa de ordenação do algoritmo Heapsort ?
  - A seleção e o posicionamento do elemento máximo são feitos em tempo constante.
  - No pior caso, a função AjustaHeap efetua $\Theta(h)$ comparações e trocas, onde h é a altura do heap que contém os elementos que restam ordenar.
  - Como o heap representa uma árvore binária completa, então $h \in \Theta(\log i)$, onde i é o número de elementos do heap na i-ésima iteração.

Transformar e Conquistar: HeapSort

- Análise: (em função da altura da árvore, h)
  - Logo, a complexidade da etapa de ordenação do Heapsort é:

$$\sum_{i=2}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n = O(n \log n)$$

  - Na verdade, $\Sigma \log i \in \Theta(n \log n)$.
  - No entanto, também temos que computar a complexidade de construção do heap

Transformar e Conquistar: HeapSort

- Mas, como construímos o heap ?
  - Se o trecho de 1 a *i* do vetor tem estrutura de heap, é fácil adicionar a folha *i + 1* ao heap e em seguida rearranjá-lo, garantindo que o trecho de 1 a *i* + 1 tem estrutura de heap
  - Esta é a abordagem top-down para construção do heap

```
Construção do Heap (top-down):
Entrada: Vetor A de n números inteiros.
Saída: Vetor A com estrutura de heap.
para i:=2 até n faça
   v := A[ i ]
   j := i
   enquanto j > 1 e A[j / 2] < v faça
      A[j ] := A[j / 2]
      j := j / 2
   A[j ] := v
```

Transformar e Conquistar: HeapSort

- Análise (comparações e trocas no pior caso):
  - O rearranjo do heap na iteração *i* efetua $\Theta(h)$ comparações e trocas no pior caso, onde *h* é a altura da árvore representada pelo trecho do heap de 1 a *i*. Logo,
    - $h \in \Theta(\log i)$
  - Portanto, o número de comparações e trocas efetuadas na construção do heap por esta abordagem é
    - $\Sigma_i \log i \in \Theta(n \log n)$

Transformar e Conquistar: HeapSort

- É possível construir o heap de forma mais eficiente.
- Suponha que o trecho de i a n do vetor é tal que, para todo j ,$i \leq j \leq n$, a sub-árvore de raiz j representada por esse trecho do vetor tem estrutura de heap.
- Note que, em particular, o trecho de $\lfloor n/2 \rfloor + 1$ a n do vetor satisfaz a propriedade, pois inclui apenas folhas da árvore binária de n elementos.

Transformar e Conquistar: HeapSort

Construção do Heap (bottom-up):

Entrada: Vetor A de *n* números inteiros.

Saída: Vetor A com estrutura de heap.

```
para i:=n/2 até 1 faça
  AjustaHeap(A,i,n)
```

Transformar e Conquistar: HeapSort

- Análise:
  - Quantas comparações são executadas no pior caso na construção do heap pela abordagem bottom-up ?
  - O rearranjo do heap na iteração *i* efetua 2*k* comparações no pior caso, onde *k* é a altura da sub-árvore de raiz *i*.
  - A altura de um nodo varia entre 1 e h=lg(n+1)-1
  - Em cada nível de altura *k* temos 2^{h-k} nodos
  - A função de custo para o número de comparações será então da forma:

$$f(n) = \sum_{k=1}^{h} 2k2^{h-k} = 2n - 2\lg(n+1)$$

  - Logo, $f(n) \in \Theta(n)$ e a abordagem bottom-up para construção do heap apenas efetua $\Theta(n)$ comparações e trocas no pior caso.

Transformar e Conquistar: HeapSort

- <u>Análise:</u>
  - Quantas comparações são executadas no pior caso na construção do heap pela abordagem bottom-up ?
  - O rearranjo do heap na iteração *i* efetua 2*k* comparações no pior caso, onde *k* é a altura da sub-árvore de raiz *i*.
  - A
  - E
  - A … a f

Assim, a complexidade do Heapsort no pior caso é Θ(n log n).

$$f(n) = \sum_{k=1} 2k2^{h-k} = 2n - 2\lg(n+1)$$

- Logo, $f(n) \in \Theta(n)$ e a abordagem bottom-up para construção do heap apenas efetua $\Theta(n)$ comparações e trocas no pior caso.

Decrementar e Conquistar

- A técnica, também chamada indutiva ou incremental, se baseia na seguinte estratégia:
  - Reduzir a instância do problema para uma instância menor do mesmo problema
  - Resolver a instância menor
  - Estender a instância menor para obter a solução para o problema original
- A técnica pode ser usada para o projeto de algoritmos recursivos e não recursivos.

Decrementar e Conquistar: Ordenação por inserção

- Problema: Ordenar um conjunto de $n \geq 1$ inteiros.
- Hipótese de Indução:
  - Sabemos ordenar um conjunto de $n-1 \geq 1$ inteiros.
- Caso base: $n = 1$
  - Um conjunto de um único elemento está ordenado.
- Passo da Indução:
  - Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S.
  - Por hipótese de indução, sabemos ordenar o conjunto $S - x$, basta então inserir x na posição correta para obtermos S ordenado.

Decrementar e Conquistar: Ordenação por inserção

```
Inserção (A, n)
---------------
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.
se n ≥ 2 faça
   Inserção (A, n - 1)
   v := A[n]
   j := n - 1
   enquanto (j > 1) e (A[j ] > v) faça
      A[j + 1] := A[j ]
      j := j - 1
   A[j + 1] := v
```

Decrementar e Conquistar: Ordenação por inserção

- É fácil eliminar o uso de recursão simulando com um laço:

```
Inserção (A)
------------
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.
para i := 2 até n faça
  v := A[i ]
  j := i -1
  enquanto (j > 1) e (A[j ] > v) faça
    A[j + 1] := A[j]
    j := j -1
  A[j + 1] := v
```

- <u>Análise</u>: Quantas comparações e quantas trocas o algoritmo executa no pior caso?

Decrementar e Conquistar: Geração de Permutações

- Gerar todas as permutações dos elementos de um vetor.
- Solução: Fixar um elemento e gerar todas as (n-1)! permutações com os elementos 2..n do vetor.
- Solução:

```
Permutação (v, i, n)

se i=n então imprima(v,n)
senão
  para k:=1 até n-i+1 faça
    Permutação (v,i+1,n)
    Rotaciona (v,i,n)
```

Decrementar e Conquistar: Geração de Permutações

- Análise em função do número de impressões:
  - Faça uma mudança de variável: $m=n-i+1$
  - Atenção na expansão telescópica

Exercício

- Escrever uma solução para o problema de encontrar a moda de uma lista, utilizando a técnica de:
  - Força bruta
  - Transformação
- Fazer a análise das soluções

"My favorite things in life don't cost any money. It's really clear that the most precious resource we all have is time."

Steve Jobs