Projeto e Análise de Algoritmos

Prof. Dr. Ednaldo B. Pizzolato

www.dc.ufscar.br/~ednaldo

ednaldo@dc.ufscar.br

TRANSFORMAÇÃO E CONQUISTA

Transformação e Conquista

- Introdução
- Gaussian elimination
  - Decomposição LU
  - Matriz inversa
  - Determinante
- Heaps e heapsort
- Regra de Horner e exponenciação binária

Transformação e Conquista

- Redução de problema
- Tarefa
- Resumo

INTRODUÇÃO

Introdução

- Esta técnica de projeto de algoritmos é baseada na ideia de transformação (por isso o nome). Na verdade são duas fases:
  - Transformação
  - Conquista

  A primeira fase é a fase em que o problema é transformado em um outro problema e a segunda fase é responsável por sua solução.

Introdução

- Existem 3 importantes variações desta técnica, todas relacionadas com a fase de transformação:
  - Simplificação de instância;
  - Mudança de representação;
  - Redução do problema (para um em que já existe um algoritmo)

Introdução

- Em muitos casos, uma das transformações aplicadas ao problema está relacionada com pré-ordenação de uma lista de elementos. Isso é fácil de se observar se, por exemplo, quisermos saber se um vetor tem elementos repetidos ou não. Claro que se ordenarmos o vetor, elementos repetidos aparecerão próximos uns dos outros. Da mesma forma, busca de elementos em um vetor pode ser mais rápida se ele estiver ordenado (por que?).

Exemplo

- Achar um elemento que repete com frequência em uma lista
- R.: Dada uma lista de valores, o método de força bruta criaria uma nova lista com elementos distintos e atualizaria a frequência do elemento encontrado (ou o colocaria na lista de elementos distintos caso lá ainda não estivesse). A lista de elementos distintos tende – com o tempo – a aumentar de tamanho.

Exemplo

- Como um elemento da lista é comparado com todos os elementos distintos encontrados até o momento, o custo da solução pertence a $\Theta(n^2)$ no pior caso.
- Se a lista estiver previamente ordenada – com custo $\Theta(n \log n)$, por exemplo – então verificar se existem repetições se transforma no problema de encontrar elementos iguais adjacentes uns dos outros (o que pode ser feito em $\Theta(n)$). Assim, $\Theta(n \log n) + \Theta(n) < \Theta(n^2)$.

ELIMINAÇÃO GAUSSIANA

Eliminação Gaussiana

- Eliminação Gaussiana é uma técnica utilizada para resolver n equações lineares com n variáveis. A forma mais comumente encontrada é com n igual a 2 ou 3.

$$a_{11}\, x + a_{12}\, y = b_1$$
$$a_{21}\, x + a_{22}\, y = b_2$$

Cabe lembrar que se os coeficientes de uma equação não são proporcionais aos da outra, o sistema tem solução única.

Eliminação Gaussiana

- Uma forma de resolver o problema é expressar uma variável em relação a outra utilizando uma das equações:

$$x = (b_1 - a_{12}\, y)/a_{11}$$

e realizar a substituição na outra equação:

$$a_{21}\,(b_1 - a_{12}\, y)/a_{11} + a_{22}\, y = b_2$$

Eliminação Gaussiana

- Na teoria, a solução (de representação e substituição) aplicada para 2 equações pode ser replicada para n equações. Na prática, tal algoritmo seria muito complexo.
- Eliminação Gaussiana é uma solução que objetiva transformar o problema em um equivalente (em que a solução é a mesma para ambos os problemas) utilizando uma matriz triangular superior.

Eliminação Gaussiana

$$
\begin{array}{ccccccccccccc}
a_{11}\,x & + & a_{12}\,x & + & a_{13}\,x & + & \ldots & + & a_{1n}\,x & = & b_1 \\
a_{21}\,x & + & a_{22}\,x & + & a_{23}\,x & + & \ldots & + & a_{2n}\,x & = & b_2 \\
\ldots & & & & & & & & & & \\
a_{n1}\,x & + & a_{n2}\,x & + & a_{n3}\,x & + & \ldots & + & a_{nn}\,x & = & b_n
\end{array}
$$

Eliminação Gaussiana

$$
\begin{array}{lllllllllll}
a_{11}\,x & + & a_{12}\,x & + & a_{13}\,x & + & \ldots & + & a_{1n}\,x & = & b_1 \\
 & + & a_{22}\,x & + & a_{23}\,x & + & \ldots & + & a_{2n}\,x & = & b_2 \\
\ldots & & & & & & & & & & \\
 & & & & & & & & a_{nn}\,x & = & b_n
\end{array}
$$

a_{11}	a_{12}	a_{13}	...	a_{1n-1}	a_{1n}
0	a_{22}	a_{23}	...	a_{2n-1}	a_{2n}
0	0	a_{33}	...	a_{3n-1}	a_{3n}
...					
0	0	0	...	0	a_{nn}

Eliminação Gaussiana

- Forward elimination

FE(A[1..n, 1..n],B[1..n])

para i ← 1 até n faça

A[i,n+1] ← B[i]

para i ← 1 até n-1 faça

para j ← i+1 até n faça

para k ← i até n+1 faça

A[j,k] ← A[j,k] – A[i,k]*A[j,i]/A[i,i]

Se A[i,i] for zero, não pode dividir. Deverá, então, trocar com outra linha... Se a solução for única deve haver tal linha!

Eliminação Gaussiana

- Forward elimination

FE(A[1..n, 1..n],B[1..n])
  para i ← 1 até n faça
    A[i,n+1] ← B[i]
  para i ← 1 até n-1 faça
    para j ← i+1 até n faça
      para k ← i até n+1 faça
        A[j,k] ← A[j,k] – A[i,k]*A[j,i]/A[i,i]

Outro problema é A[i,i] ser tão pequeno que cause distorção no resultado da divisão.

Eliminação Gaussiana

- Forward elimination

```
FE(A[1..n, 1..n],B[1..n])
   para i ← 1 até n faça
        A[i,n+1] ← B[i]
   para i ← 1 até n-1 faça
        para j ← i+1 até n faça
            para k ← i até n+1 faça
                A[j,k] ← A[j,k] – A[i,k]*A[j,i]/A[i,i]
```

De qualquer forma, temos 3 loops. Isso nos indica que o algoritmo é $\Theta(n^3)$

DECOMPOSIÇÃO LU

Decomposição LU

- Como resultado indireto da eliminação Gaussiana tem-se a decomposição LU. Na verdade, as implementações atuais já contemplam a decomposição LU no lugar do algoritmo explicitado anteriormente.
- O conceito é obter 2 matrizes, uma superior (U) e outra inferior (L), de tal forma que o produto LxU seja igual à matriz original.

Decomposição LU

1	0	0
2	1	0
1/2	1/2	1

×

2	-1	1
0	3	-3
0	0	2

=

2	-1	1
4	1	-1
1	1	1

L

U

Decomposição LU

- Resolver um sistema Ax = b é equivalente a resolver o systema LUx = b. Se considerarmos que y = Ux, então o sistema original será Ly=b.
- Resolver o sistema Ly=b é bem mais fácil porque a matriz é triangular inferior. Depois, com y é possível obter x (tendo U como triangular superior).
- Dependendo da implementação é possível não ter nem que utilizar memória extra, pois as duas matrizes (L e U) podem ser armazenadas em uma só.

MATRIZ INVERSA

Matriz inversa

- A matriz A^{-1} (n x n) inversa de A (também n x n) é aquela que, quando multiplicada pela matriz A produz a matriz identidade (1s na diagonal e 0s no resto).

$$A.A^{-1} = I$$

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Matriz inversa

- É importante observar que nem todas as matrizes possuem inversas. Se uma matriz A não possui inversa, então ela é chamada de matriz singular.
- Se ela possui inversa, então a inversa será única.
- Uma maneira de se verificar se a matriz não é singular é aplicando a Eliminação Gaussiana: se for possível obter uma matriz triangular superior com elementos da diagonal diferentes de 0, então ela não é singular; caso contrário é singular.

Matriz inversa

- Assim como podemos transformar a expressão $a.x = b$ em $x = a^{-1}.b$ (quando a não é zero), podemos fazer o mesmo com um sistema de equações:

$$A.x = b$$

produzindo:

$$x = A^{-1}.b \text{ (se A não for singular)}$$

DETERMINANTE

Determinante

- Também é possível utilizar a Eliminação Gaussiana para calcular o determinante de uma matriz.
- O determinante de uma matriz A_{nxn} é representado por |A| e pode ser calculado de forma recursiva da seguinte forma:
  - Se n = 1, então o determinante é $a_{1,1}$
  - Se for maior que um, então será obtido pela fórmula:

$$|A| = \sum_{j=1}^{n} s_j a_{1,j} |A_j|$$

Determinante

$$|A| = \sum_{j=1}^{n} s_j a_{1,j} |A_j|$$

Onde s_j é +1 (se j for ímpar) ou -1 (se j for par);
$a_{1,j}$ é o elemento da linha 1 e coluna j;
$|A_j|$ é o determinante da matriz obtida pela exclusão da linha 1 e coluna j.

Determinante

- E se desejarmos obter o determinante de uma matriz muito grande? Usando a fórmula recursiva chegamos ao cálculo de n! termos.
- Se utilizarmos a Eliminação Gaussiana conseguimos obter um valor que é o determinante ou está relacionado a ele pela troca de um sinal ou pela multiplicação de uma constante. O truque é fazer o cálculo do determinante da matriz triangular superior.

Determinante

- E qual a vantagem de se fazer isso?

Determinante

- E qual a vantagem de se fazer isso?
- O cálculo pode ser feito em $O(n^3)$!

Transformação e Conquista

- Introdução
- Gaussian elimination
  - Decomposição LU
  - Matriz inversa
  - Determinante
- Heaps e heapsort
- Regra de Horner e exponenciação binária

Heaps e Heapsort

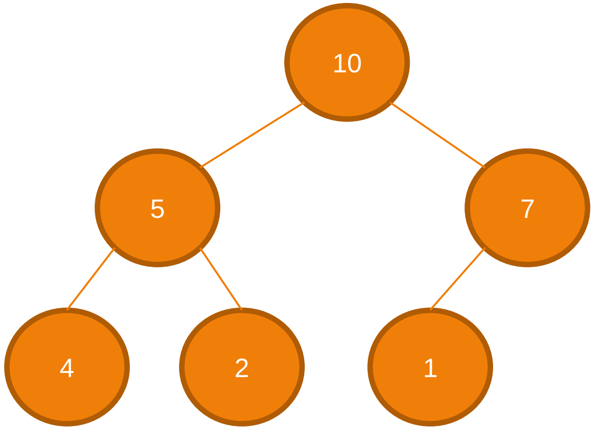
- Heaps são pilhas parcialmente ordenadas que servem para implementar filas com prioridades.
- Existem diversos algoritmos que utilizam heaps, como o algoritmo de Huffman e o de Dijkstra que serão explorados mais a fundo na aula de algoritmos gulosos (greedy).
- Heap também serve como pilar para a implementação do algoritmo de ordenação chamado Heapsort.

Heaps

- Definição: Uma heap pode ser definida como uma árvore binária com chaves nos nós (uma chave por nó) considerando-se que as duas condições a seguir sejam satisfeitas:
  - Propriedade de forma: uma árvore binária é essencialmente completa (ou simplesmente completa) quando todos os seus níveis estão cheios exceto, possivelmente, o último nível, quando alguns elementos do lado direito podem não estar presentes.

Heaps

- Propriedade Heap: a chave em cada nó é maior ou igual às chaves dos filhos.



Heaps

- Como construir uma heap a partir de uma lista de chaves?

Heaps

Algoritmo HeapBottomUp(H[1..n])

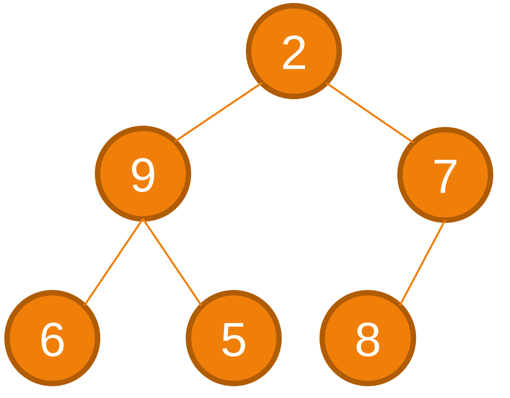
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```

2	9	7	6	5	8

Heaps

Algoritmo HeapBottomUp(H[1..n])

```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```

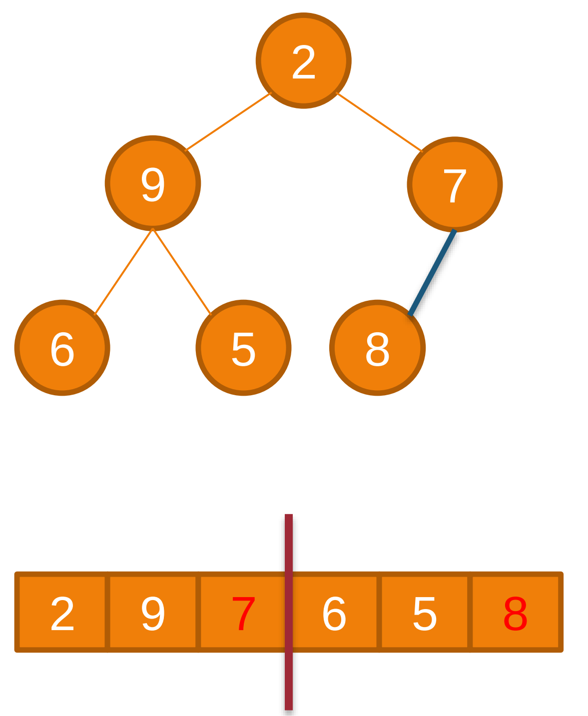


2	9	7	6	5	8

Heaps

Algoritmo HeapBottomUp(H[1..n])

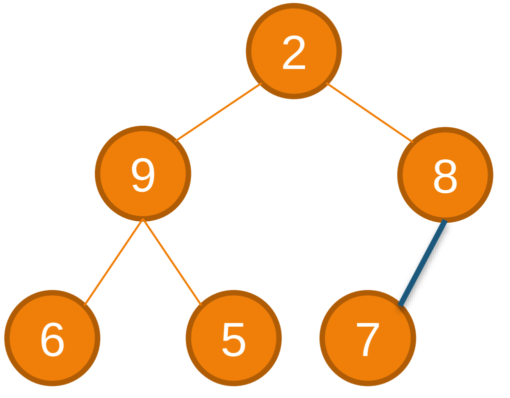
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```

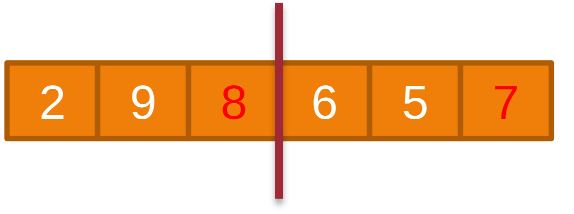


Heaps

Algoritmo HeapBottomUp(H[1..n])

```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```

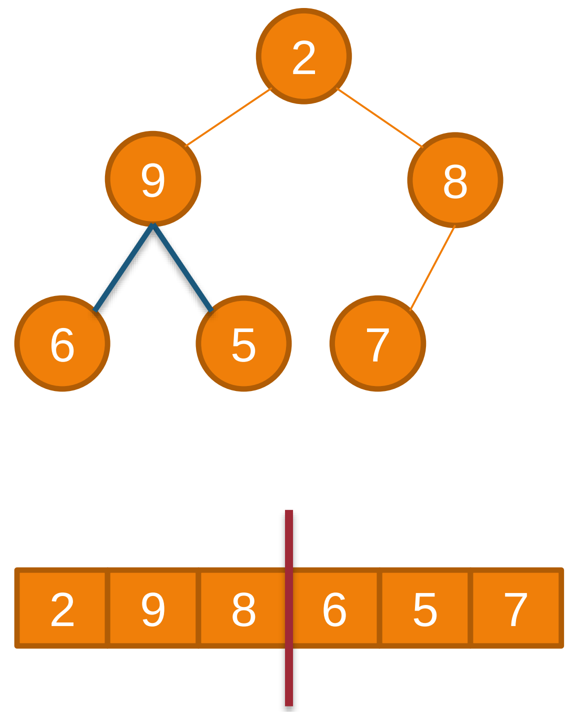




Heaps

Algoritmo HeapBottomUp(H[1..n])

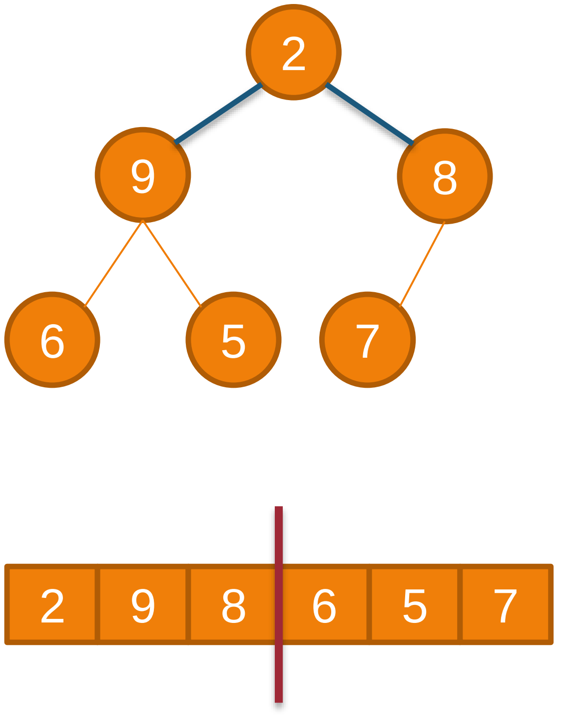
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```



Heaps

Algoritmo HeapBottomUp(H[1..n])

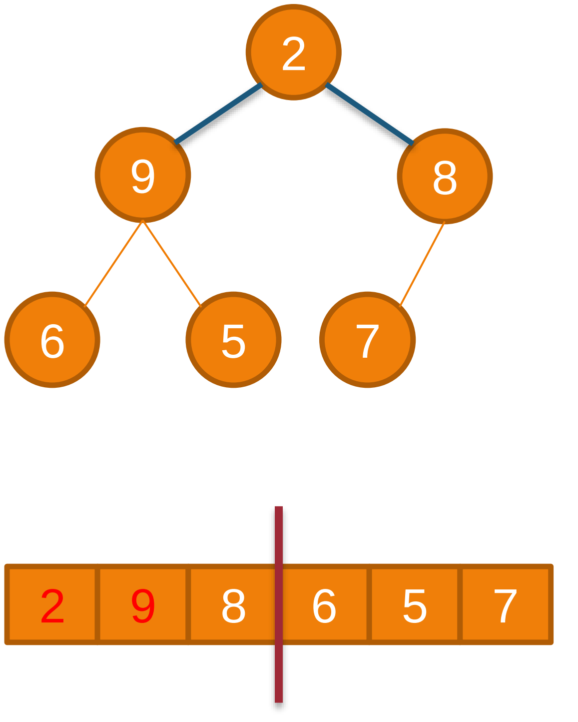
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```



Heaps

Algoritmo HeapBottomUp(H[1..n])

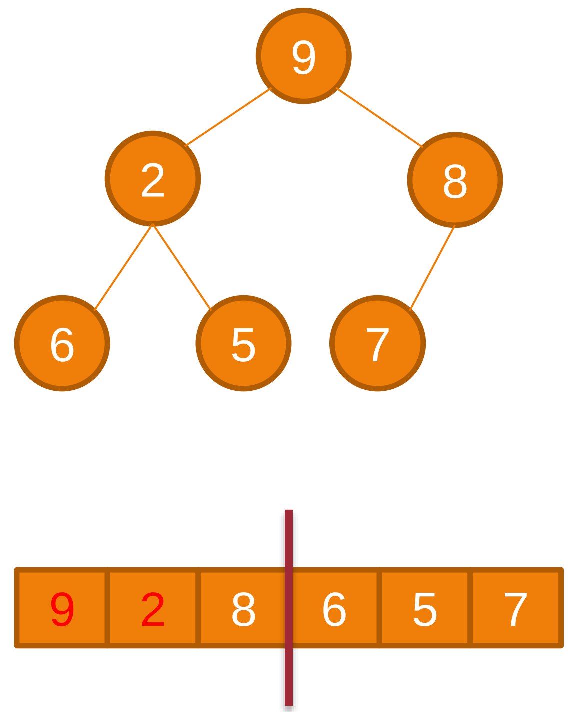
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```



Heaps

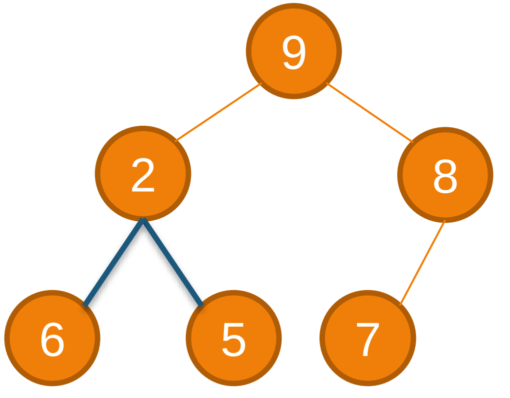
Algoritmo HeapBottomUp(H[1..n])

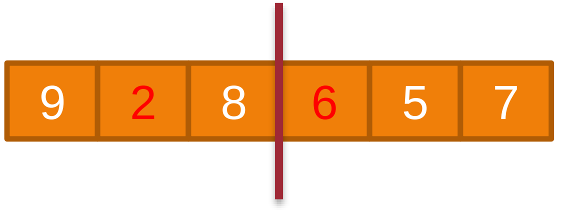
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```



Algoritmo HeapBottomUp(H[1..n])

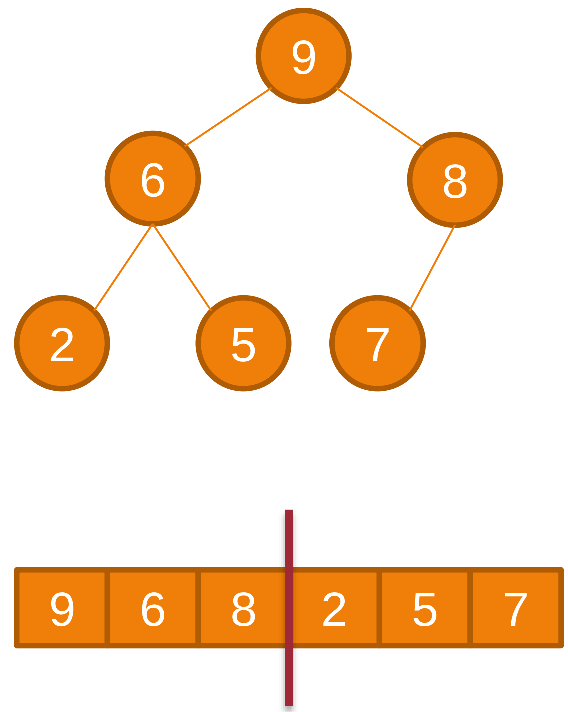
```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```





Algoritmo HeapBottomUp(H[1..n])

```
para i ←⌊n/2⌋ até 1 faça
    k ←i; v ←H[k]
    heap ←falso
    enquanto não heap E 2*k ≤ n faça
        j ← 2*k
        se j < n então // existem 2 filhos
            se H[j] < H[j+1] então
                j ← j + 1
        se v ≥ H[j] então
            heap ← verdadeiro
    H[k] ← v
```



HEAPSORT

Heapsort

- Primeiro precisamos saber como retirar o maior elemento da Heap
- Retirada do maior elemento de uma Heap
  - Passo 1: trocar a raiz com a última chave da Heap
  - Passo 2: diminua o tamanho da heap em 1 unidade
  - Passo 3: heapficar a árvore

Heapsort

- O algoritmo Heapsort possui 2 fases:
  - Fase 1: criar uma heap dado um vetor
  - Fase 2: Aplicar o algoritmo de eliminação da raiz n-1 vezes.

Como a eliminação é feita pelo maior elemento (que é colocado no final do vetor), na verdade o que teremos, no final do processo, um vetor ordenado.

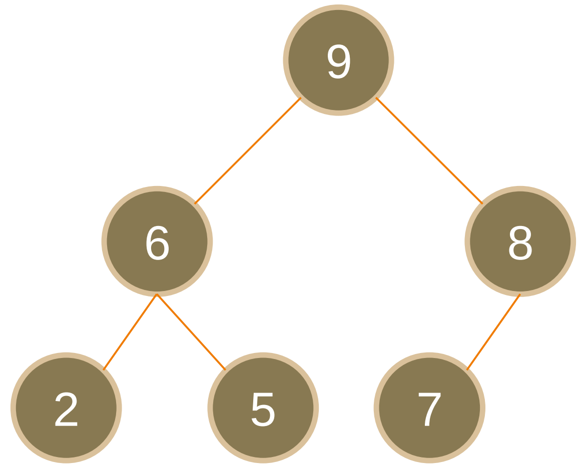
Heapsort

- Exemplo:

2	9	7	6	5	8

- Fase 1:

9	6	8	2	5	7



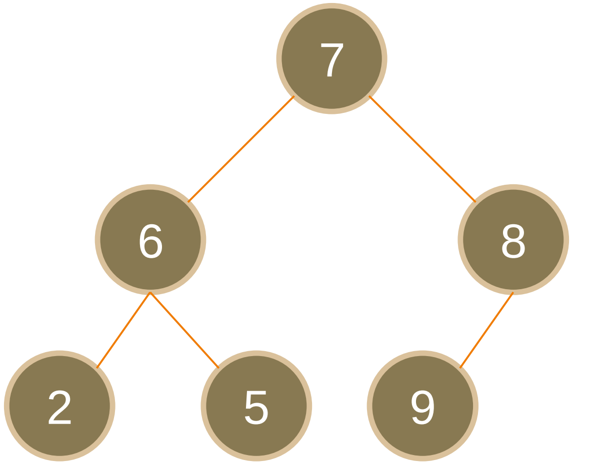
Heapsort

- Resultado da fase 1:

9	6	8	2	5	7

- Fase 2:

7	6	8	2	5	9



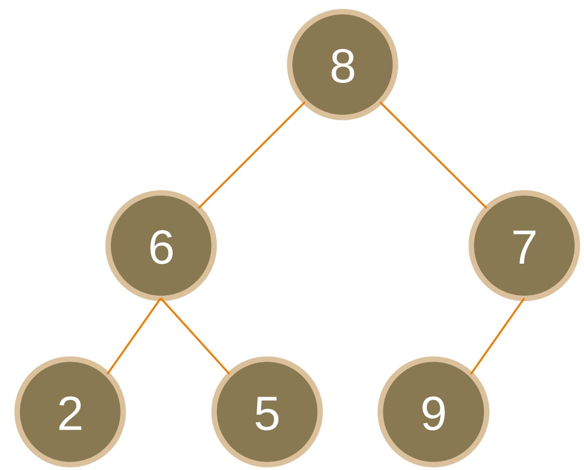
Heapsort

- Resultado da fase 1:

9	6	8	2	5	7

- Fase 2:

8	6	7	2	5	9



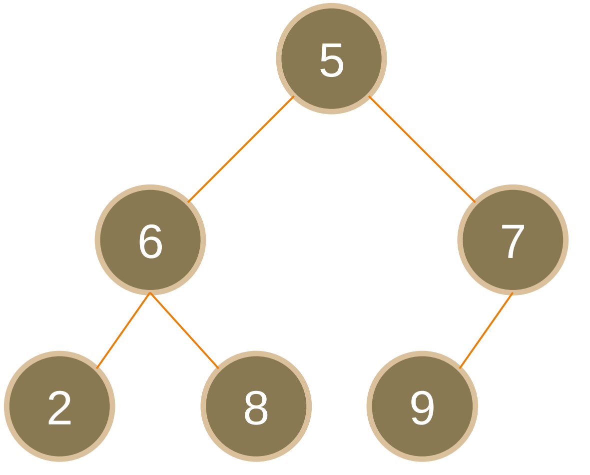
Heapsort

- Resultado do passo anterior:

8	6	7	2	5	9

- Fase 2:

5	6	7	2	8	9



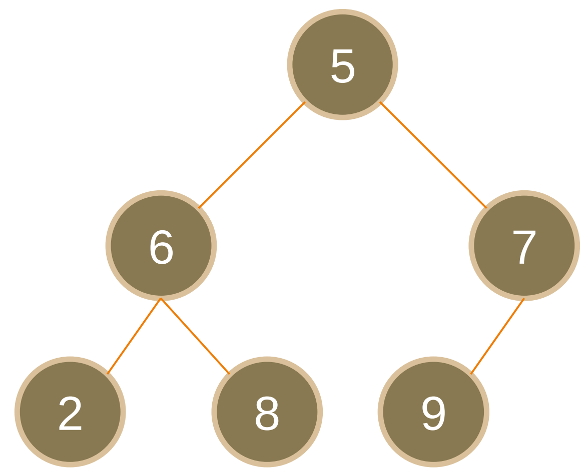
Heapsort

- Resultado do passo anterior:

8	6	7	2	5	9

- Fase 2:

5	6	7	2	8	9



Heapsort

- Resultado do passo anterior:

8	6	7	2	5	9

- Fase 2:

7	6	5	2	8	9

Transformação e Conquista

- Introdução
- Gaussian elimination
  - Decomposição LU
  - Matriz inversa
  - Determinante
- Heaps e heapsort
- Regra de Horner e exponenciação binária

REGRA DE HORNER

Regra de Horner

- A regra de Horner serve para calcular de forma eficiente um polinômio da forma:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

E ter uma forma eficiente de calcular tal polinômio pode servir, por exemplo, para calcular a transformada rápida de Fourier (FFT).

Regra de Horner

- Mais uma vez, o conceito está em transformar a forma como se resolve. Ao invés de se utilizar a fórmula eu seu formato original, pode-se utilizar a representação em que o x é colocado sucessivamente em evidência:

$$p(x) = (...(a_n x + a_{n-1})x + ...)x + a_0$$

Regra de Horner

- Vejamos um exemplo:

$$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$$
$$= x(2x^3 - x^2 + 3x + 1) - 5$$

$$= x(x(x(2x - 1) + 3) + 1) - 5$$

Regra de Horner

- A eficiência do algoritmo de Horner é muito elevada. Observe que ele requer n multiplicações, enquanto que se utilizarmos força bruta gastaremos n multiplicações apenas com o termo $a_n x^n$.

```
Algoritmo Horner(P[0..n],x)
    poli ← P[n]
    para i ← n-1 até 0 faça
        poli ← x * poli + P[i]
    retorna poli
```

Regra de Horner

coeficientes	2	-1	3	1	-5
x = 3	2	3.2+(-1)=5	3.5+3=18	3.18+1=55	3.55+(-5) = 160

$$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$$

Transformação e Conquista

- Introdução
- Gaussian elimination
  - Decomposição LU
  - Matriz inversa
  - Determinante
- Heaps e heapsort
- Regra de Horner e exponenciação binária

EXPONENCIAÇÃO BINÁRIA

Exponenciação binária

- De forma quase que inacreditável o algoritmo de Horner degenera para força bruta quando se precisa calcular a^n (ou seja, x^n quando $x = a$).
- É possível melhorar este desempenho se tratarmos o expoente n como um número binário:

$$n = b_l \ldots b_i \ldots b_0$$

Mas como isso pode ser útil?

Exponenciação binária

Simples! Basta considerarmos a expressão x^n como sendo:

$$p(x) = b_I x^I + \ldots + b_i x^i + \ldots + b_0$$

Assim, se x = 2 e n = 13 (1101) teríamos:

$$a^n = a^{p(2)} = a^{b_I 2^I + \ldots + b_i 2^i + \ldots + b_0}$$

Exponenciação binária

- A “moral da história” é que se o b_i for 0 então o valor é elevado ao quadrado; se for 1 deve-se ainda fazer uma multiplicação extra:

```
função exp(a,b(n))
produto ← a
para i ← l -1 até 0 faça
        produto ← produto*produto
        se bi = 1 então
                produto ← produto*a
retorna produto
```

Exponenciação binária

- Novamente houve uma transformação do problema para se obter uma solução mais eficiente.
- Em geral teremos (b-1) ≤ M(n) ≤ 2(b-1)
- Como b é o tamanho da sequência de bits que representa n, temos:

$$b - 1 = \lfloor \log_2 n \rfloor$$

Ou seja, a eficiência passou a ser logarítmica!

Transformação e Conquista

- Introdução
- Gaussian elimination
  - Decomposição LU
  - Matriz inversa
  - Determinante
- Heaps e heapsort
- Regra de Horner e exponenciação binária

Transformação e Conquista

- Redução de problema
- Tarefa
- Resumo

REDUÇÃO DE PROBLEMA

Redução de problema

- Se temos um problema X que pode ser reduzido (modificado, transformado) em um problema Y e o problema Y tem uma solução conhecida, então temos uma redução de problema.

Exemplo: Mínimo Multiplo Comum (MMC)

O MMC de 2 números inteiros positivos m e n, denotado por MMC(m,n) é definido como sendo o menor inteiro que é divisível por m e por n. Exemplo: MMC(12,60) = 120 e MMC(11,5) = 55.

Redução de problema

- O algoritmo conhecido é o da multiplicação dos primos comuns:

  24 = 2.2.2.3   (24/2 = 12; 12/2 = 6; 6/2 = 3; 3/3 = 1)

  60 = 2.2.3.5   (60/2 = 30; 30/2 = 15; 15/3 = 5; 5/5 = 1)

Redução de problema

Este algoritmo é bem ineficiente. Um algoritmo bem mais eficiente é reduzir o problema para:

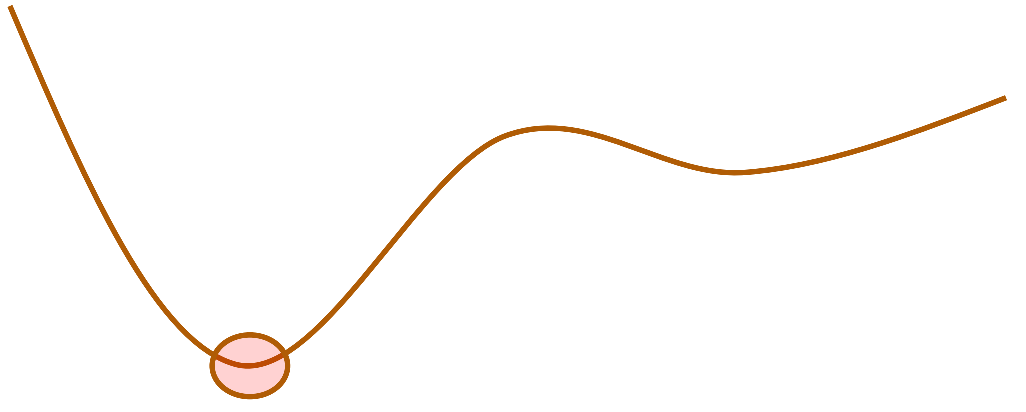
$$MMC(m,n) = \frac{m.n}{MDC(m,n)}$$

E, utilizando o algoritmo de Euclides, é possível solucionar o denominador de forma bem eficiente!

Problemas de otimização

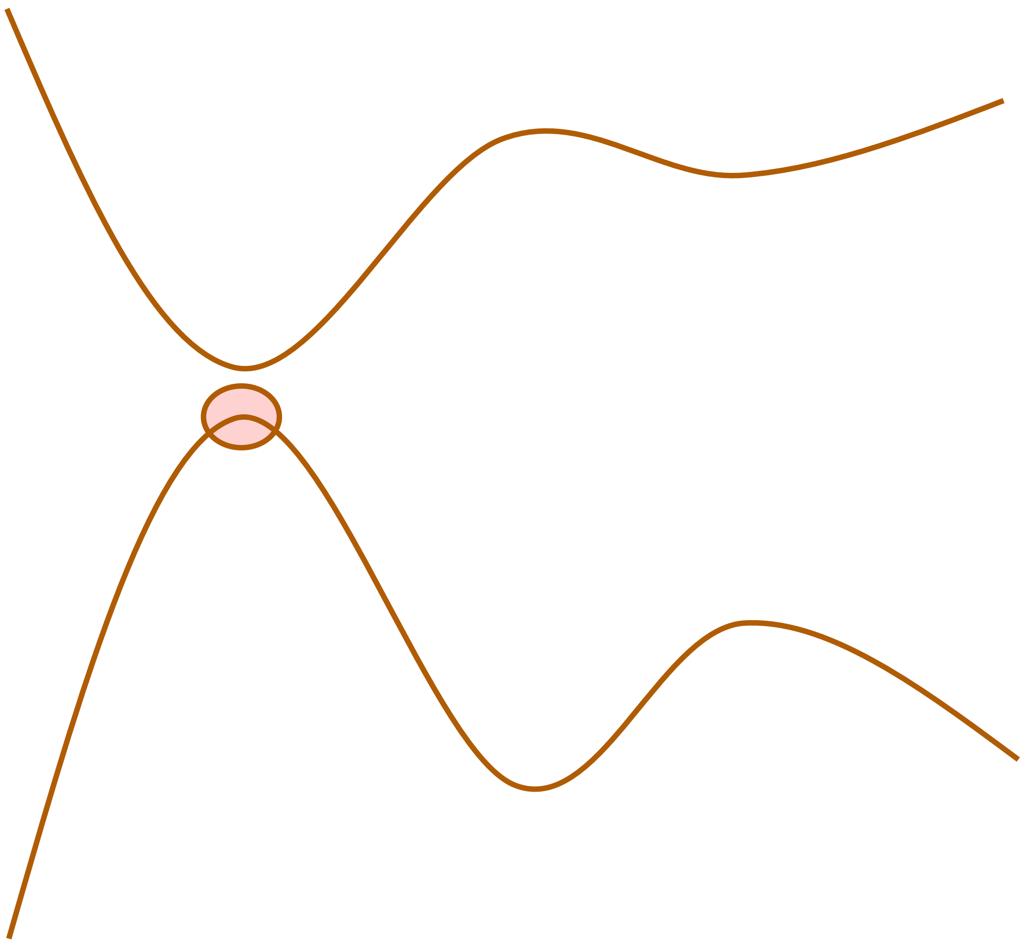
- Em problemas de otimização é possível que se queira achar o máximo (problema de maximização) ou o mínimo (problema de minimização).

Problemas de otimização



- Suponha que você tenha um algoritmo para achar o máximo de uma função, mas o que deseja é achar o mínimo. O que fazer?

Problemas de otimização



Achar o mínimo de uma função é o mesmo que achar o máximo da inversa da função com sinal trocado.

Redução para problemas de programação linear

- A programação linear trata de otimização de funções lineares sujeitas a várias restrições.
- Exemplo: Pense em alguém que queira fazer um investimento de R$ 100.000,00; o valor deve ser investido em ações, títulos do tesouro e poupança. O que se espera é um retorno anual de 10%, 7% e 3% respectivamente. Mas como o investimento em ações é mais arriscado, as restrições impostas pelo investidor são: a) investimentos em ações não podem ser superiores a 1/3 do investimento em títulos do tesouro; no mínimo 25% do valor investido tanto em ações e em títulos deve ser investido em poupança.

Redução para problemas de programação linear

$$F = 0.10x + 0.07y + 0.03z$$

Restrições:

$$x + y + z = 100.000,00$$

$$x \leq \frac{1}{3} y$$

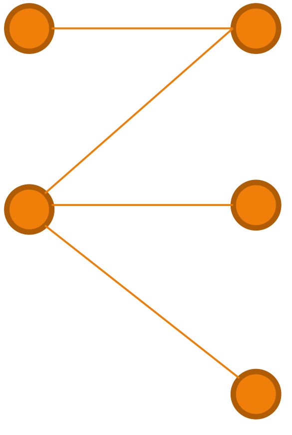
$$z \geq 0.25 (x + y)$$

$$x \geq 0, y \geq 0, z \geq 0$$

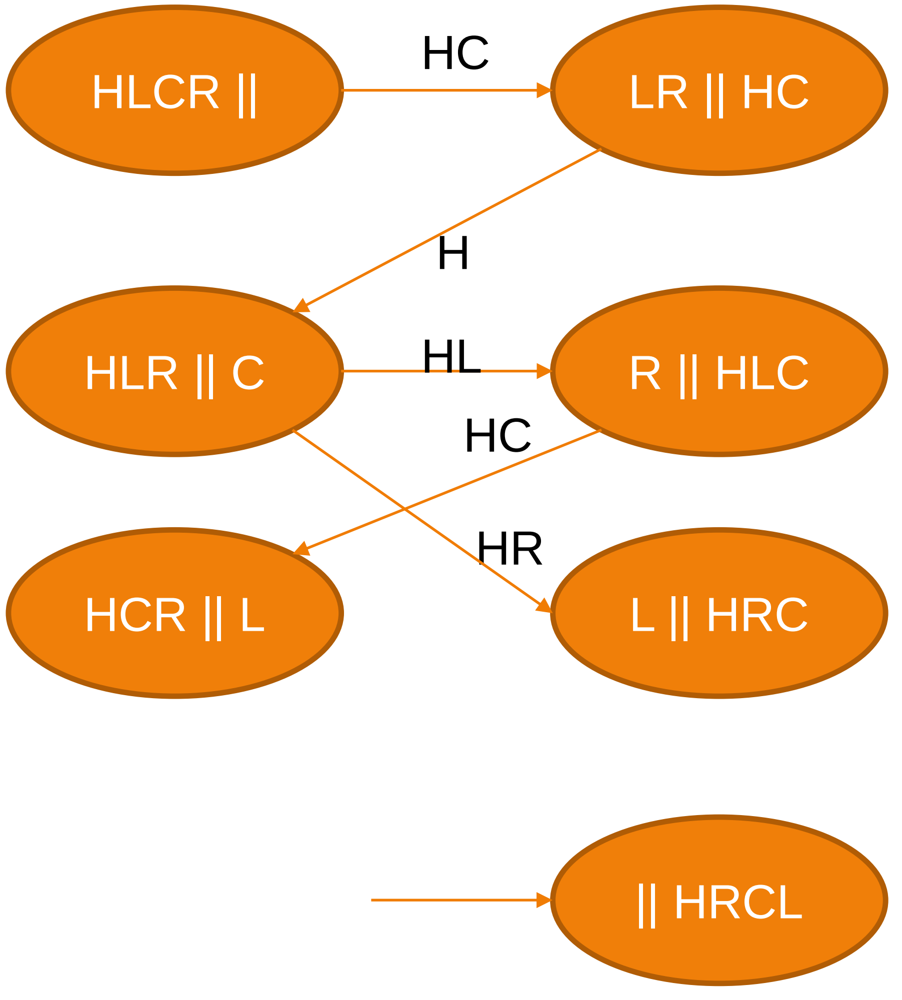
➔ A solução deste problema contempla o uso do método simplex.

Redução para problemas de grafos

- Em várias situações também é possível reduzir o problema a um problema de grafo.



Redução para problemas de grafos



Homem do campo
Lobo
Cabra
Repolho
|| rio

Transformação e Conquista

- Redução de problema
- Tarefa
- Resumo

TAREFA

Tarefa

- Pesquise um problema relacionado com a teoria apresentada em sala de aula e mostre como resolvê-lo. Você deverá postar o resultado de sua pesquisa no moodle.

Transformação e Conquista

- Redução de problema
- Tarefa
- Resumo

Resumo

- Existem 3 principais variações da estratégia de transformar e conquistar:
  - Simplificação;
  - Mudança de representação;
  - Redução de problema.

Resumo

- A simplificação compreende a transformação do problema em uma instância do mesmo problema que tem alguma propriedade especial que faz com que o problema fique mais simples de ser resolvido. Exemplos são : pré-ordenação de uma lista, Eliminação Gaussiana e rotações na AVL.

Resumo

- A Mudança de representação implica mudar a representação do problema em outra de mesma instância. Alguns exemplos são: Heaps, Heapsort, Regra de Horner e resolução de polinômio.

Resumo

- A Redução de problema é uma das variações que contempla a utilização de outro problema como parte da solução. Em algumas situações o problema pode ser reduzido a programação linear ou a um problema de grafo.

THE END