Fontes principais

1. Sedgewick, Algorithms in C, 3rd edition, Parts 1–4
   - ▷ Estruturas de dados, ordenação e busca
2. Feofiloff: Projeto de Algoritmos
   `http://www.ime.usp.br/~pf/algoritmos/`

Códigos

```
This code is from "Algorithms in C, Third Edition,"
by Robert Sedgewick, Addison Wesley Longman, 1998.
```

Problema da conexidade

```
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$ cat prog1.1.in
3 4
4 9
8 0
2 3
5 6
2 9
5 9
7 3
4 8
5 6
0 2
6 1
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$
```

Problema da conexidade

```
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$ prog1.1 < prog1.1.in
 3 4
 4 9
 8 0
 2 3
 5 6
 5 9
 7 3
 4 8
 6 1
yoshi@erdos:~/Main/www/2006ii/mac122a/exx$
```

Programas para o problema da conexidade

```
/* prog1.1.c - Quick find */
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d\n", &p, &q) == 2)
      {
        if (id[p] == id[q]) continue;
        for (t = id[p], i = 0; i < N; i++)
          if (id[i] == t) id[i] = id[q];
        printf(" %d %d\n", p, q);
      }
 }
```

Quick find

Propriedade 1. *Suponha que executamos o algoritmo quick find em uma instância com M pares e N objetos, e que a saída tem S pares. Então o algoritmo executou pelo menos NS instruções (por exemplo, ele executou o teste* `id[i] == t` *pelo menos este número de vezes).*

▷ Note que S pode chegar a ser $N - 1$. Assim, quick find pode ter tempo de execução tão grande quanto $N(N - 1)$ unidades.

```
/* prog1.2.c - Quick union */
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d\n", &p, &q) == 2) {
      for (i = p; i != id[i]; i = id[i]) ;
      for (j = q; j != id[j]; j = id[j]) ;
      if (i == j) continue;
      id[i] = j;
      printf(" %d %d\n", p, q);
    }
  }
```

Quick union

Propriedade 2. *Suponha que $M > N$. O algoritmo quick union pode chegar a executar $(N-1)(N-2)/2$ instruções para resolver o problema da conexidade com $N-1$ pares e N objetos.*

▷ Para verificar a propriedade acima, considere a entrada cujos pares são $(0,1),(0,2),\ldots,(0,N-1)$. Conte o número de vezes que a instrução `i = id[i]` (no `for`) é executada.

```
/* prog1.3.c - Weighted quick union */
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, id[N], sz[N];
    for (i = 0; i < N; i++)
      { id[i] = i; sz[i] = 1; }
    while (scanf("%d %d\n", &p, &q) == 2)
      { for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf(" %d %d\n", p, q);
      }
  }
```

Quick union

Propriedade 3 (Custo do find)**.** *O algoritmo quick union com pesos segue* $\leq \log_2 N$ *ponteiros para chegar à raiz da árvore que contém o elemento de partida.*

Prova. Basta provar o seguinte fato sobre as árvores que ocorrem em nossa estrutura de dados: se uma árvore tem k elementos, então ela tem altura no máximo $\log_2 k$. Suponha o fato verdadeiro em um dado momento. Suponha que unimos duas árvores por uma operação de *union*. Suponha que as árvores tinham i e i' elementos, com $i \leq i'$, e alturas h e h'. Se $h < h'$, então a nova árvore tem altura h'. Se $h = h'$, então a nova árvore tem altura $h + 1 \leq \log_2 i + 1 = \log_2(i + i) \leq \log_2(i + i')$. □

Quick union

Corolário 4. *O algoritmo quick union com pesos demora tempo não mais que proporcional a $M \log N$ para resolver o problema da conexidade com M pares e N objetos.*

- ▷ **Dizemos:** a complexidade de tempo do quick union com pesos é $O(M \log N)$.

Quick union com compressão de caminhos

É possível melhorar o quick union com pesos, usando *compressão de caminhos*, como no programa a seguir. [Não chegamos lá na aula.]

```
/* prog1.4.c - Weighted quick union with path compression */
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, id[N], sz[N];
    for (i = 0; i < N; i++) { id[i] = i; sz[i] = 1; }
    while (scanf("%d %d\n", &p, &q) == 2)
      { for (i = p; i != id[i]; )
          { int t = i; i = id[id[t]]; id[t] = i; }
        for (j = q; j != id[j]; )
          { int t = j; j = id[id[t]]; id[t] = j; }
        if (i == j) continue;
        if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf(" %d %d\n", p, q);
      }
  }
```

Exercício

▷ **Instância conexa:** dizemos que uma instância é conexa quando a saída tem $N - 1$ pares.

Para contar o número de pares na saída, basta fazer

```
yoshi@RANDOM ~/Main/www/2006ii/mac122a/exx
$ prog1.1 < prog1.1.in | wc -l
9
yoshi@RANDOM ~/Main/www/2006ii/mac122a/exx
$
```

Verifique experimentalmente a *probabilidade de uma instância aleatória ser conexa* para valores grandes de N e M por volta de $N(\log N)/2$. (Você tem de escrever um pequeno programa para gerar as instâncias aleatórias.)