Algoritmos e Estruturas de Dados

- **Algoritmos**: métodos para a resolução de problemas [passíveis de implementação em computador]
- **Estruturas de Dados**: forma ou processo de guardar informação

Algoritmos eficientes usam "boas" estruturas de dados

- Objectivo da disciplina é estudar um número variado e significativo de algoritmos
  - áreas de aplicação relevantes mas diversas
  - algoritmos básicos
  - compreender e apreciar o seu funcionamento e a sua relevância

ex: ordenação, procura, grafos, etc

Algoritmos

- Estratégia:
  - especificar (definir propriedades)
  - arquitectura (algoritmo e estruturas de dados)
  - implementar (numa linguagem de programação)
  - testar (submeter entradas e verificar observância das propriedades especificadas)
    - fazer várias experiências
- Estudar e prever o desempenho e características dos algoritmos
  - possibilita desenvolvimento de novas versões
  - comparar diferentes algoritmos para o mesmo problema
  - prever ou garantir desempenho em situações reais

Algoritmos como área de estudo e investigação

- Suficientemente antiga que os conceitos básicos e a informação essencial são bem conhecidos
- Suficientemente nova de forma a permitir novas descobertas e novos resultados
- Imensas áreas de aplicação:
  - ciência
  - engenharia
  - comerciais

Relação entre Algoritmos e Estruturas de Dados

- Algoritmos são métodos para resolver problemas
  - problemas têm dados
  - dados são tratados (neste contexto) computacionalmente
- O Processo de organização dos dados pode determinar a eficiência do algoritmo
  - algoritmos simples podem requerer estruturas de dados complexas
  - algoritmos complexos podem requerer estruturas de dados simples

Para **maximizar a eficiência** de um algoritmo as estruturas de dados que se usam têm de ser projectadas em simultâneo com o desenvolvimento do algoritmo

Porquê estudar algoritmos?

- Quanto se utilizam meios computacionais:
  - queremos sempre que a computação seja mais rápida
  - queremos ter a possibilidade de processar mais dados
  - queremos algo que seria impossível sem o auxílio de um computador
- O avanço tecnológico joga a nosso favor:
  - máquinas mais rápidas, com maior capacidade, mas
  - a tecnologia apenas melhora o desempenho por um factor constante
  - bons algoritmos podem fazer muito melhor!
  - um mau algoritmo num super-computador pode ter um pior desempenho que um bom algoritmo numa calculadora de bolso

Utilização de algoritmos

- Para problemas de pequeno porte a escolha do algoritmo não é relevante
  - desde que esteja correcto
    - funcione de acordo com as especificações
- Para problemas de grandes dimensões (ou com grande volume de dados)
  - obviamente tem de estar correcto
  - a motivação é maior para optimizar o tempo de execução e o espaço (memória, disco) utilizado
  - o potencial é enorme
    - não apenas em termos de melhoria de desempenho
    - mas de podermos fazer algo que seria impossível de outra forma

Tarefas na resolução de um problema

- Compreensão e definição do problema a resolver
  - completa e sem ambiguidades
- Gestão da complexidade
  - primeiro passo na resolução do problema
- Sempre que possível,
  - decompor o problema em sub-problemas de menor dimensão/complexidade
    - por vezes a resolução de cada sub-problema é trivial
    - importante ter completo domínio de algoritmos básicos
    - reutilização de código (aumento de produtividade!)

Escolha de um algoritmo para um dado problema [1]

- **Importante**: evitar supra-optimização
  - se um sub-problema é simples e relativamente insignificante, porquê dispender muito esforço na sua resolução
- Utilizar um algoritmo simples e conhecido
  - focar o esforço nos problemas complexos e de maior relevância para a resolução do problema (os que consomem mais recursos)

Escolha de um algoritmo para um dado problema [2]

- Escolha do melhor algoritmo é um processo complicado
  - pode envolver uma análise sofisticada
    - utilizando complexas ferramentas e/ou raciocínio matemáticas
    - Ramo da Ciência da Computação que trata do estudo deste tipo de questões é a Análise de Algoritmos
  - muitos dos algoritmos que estudaremos foram analisados e demonstrou-se terem excelente desempenho
  - importante conhecer as características fundamentais de um algoritmo
    - os recursos que necessita
    - como se compara com outras alternativas possíveis

Problema: Conectividade [1]

Dada uma sequência de pares de inteiros, ***(p, q)*** assumimos que

- cada inteiro representa um objecto de um dado tipo
- ***(p,q)*** significa que ***p*** está ligado a ***q***
- conectividade é uma relação transitiva
  - se existirem as ligações ***(p,q)*** e ***(q,r)***, então a ligação ***(p,r)*** também existe

➔ Escrever um programa para filtrar informação redundante

Entrada: pares de inteiros

Saída: pares de inteiros contendo informação nova em relação à anteriormente adquirida

Problema: Conectividade [2]

- Exemplos de aplicação:
  - definir se novas ligação entre computadores numa rede local são necessárias
  - definir se novas ligação são precisas num circuito eléctrico para assegurar a conectividade pretendida
  - definir o número mínimo de pré-requisitos para uma dada tarefa (por exemplo para uma disciplina)
  - etc, etc

Problema: Conectividade [3]

Colunas:

1 - Dados de entrada

2 - conjunto não redundante de ligações

3 - caminho usado para justificar redundância

3-4	3-4	
4-9	4-9	
8-0	8-0	
2-3	2-3	
5-6	5-6	
2-9		2-3-4-9
5-9	5-9	
7-3	7-3	
4-8	4-8	
5-6		5-6
0-2		0-8-4-3-2
6-1	6-1	

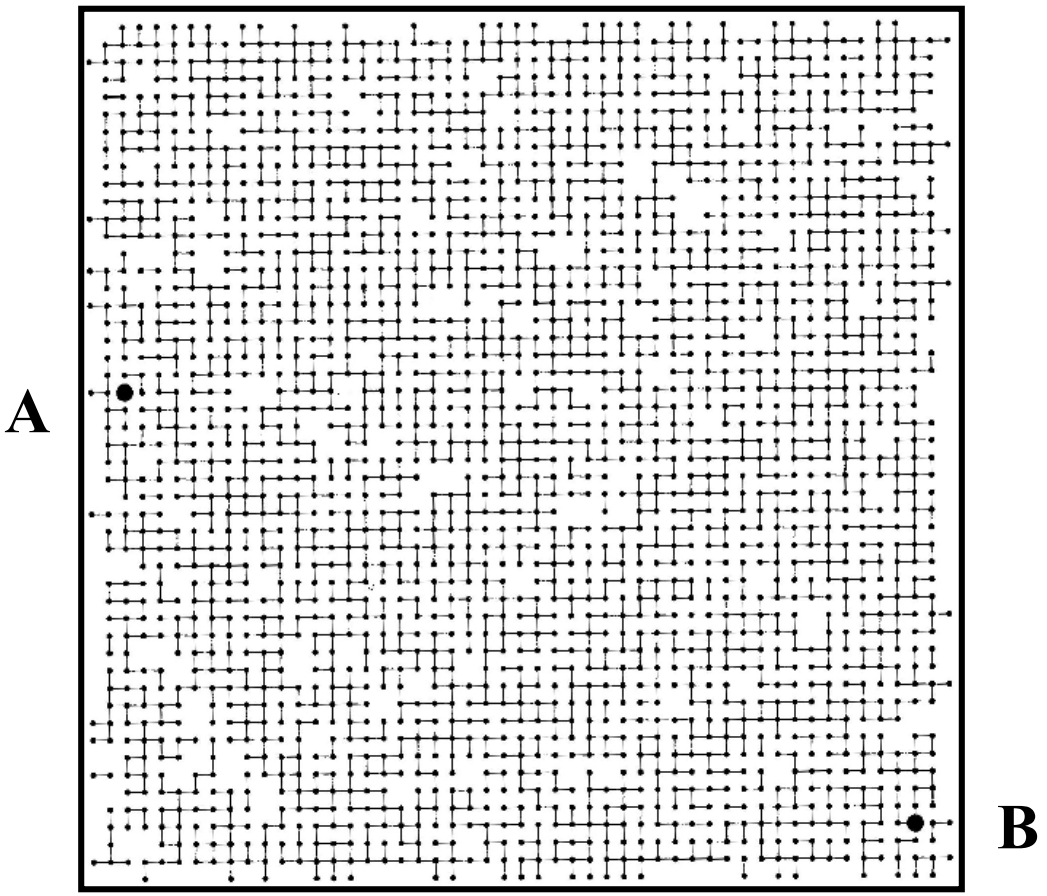
Problema: Conectividade [4]

- Porquê escrever um programa?
  - número de dados (inteiros e relações) pode ser enorme para uma análise não automática
    - Rede telefónica (em Lisboa são 2 milhões de utilizadores)
    - Circuito integrado de grande densidade (dezenas de milhões de transistores)
- Importante perceber a especificação do problema:
  - dado ***(p,q)*** programa deve saber se ***p*** e ***q*** estão ligados
  - não é **preciso** demonstrar como é que os pares estão ligados
    - por que sequência ou sequências

Modificar uma especificação pode aumentar ou reduzir significativamente a complexidade de uma problema

Problema: Conectividade [5] Visualização

A e **B** estão ligados?
Como?



Problema: Conectividade [6]
Estratégia

- Especificação:
  - definição (exacta) do problema
- Identificação da operações fundamentais a executar
  - dado um par, ver se representa uma nova ligação (*procura*)
  - incorporar novos dados se for o caso
- Implementação como operadores abstractos
  - valores de entrada representam elementos em conjuntos abstractos
  - projectar algoritmo e estrutura de dados que permita
    - procurar o conjunto que contenha um dado elemento
    - substituir dois desses conjuntos pela sua união

Generalidade na concepção pode tornar o(s) algoritmo(s) reutilizável noutro contexto

Problema: Conectividade [7]
Solução Conceptual

- Operadores *procura* (*find*) e *união* (*union*):
  - depois de ler um novo par ***(p,q)*** faz-se uma procura para cada membro do par pelo conjunto que o contém
  - se estão ambos no mesmo conjunto passamos ao par seguinte
  - se não estão calculamos a união dos dois conjuntos e passamos ao par seguinte
- Resolução do problema foi efectivamente simplificada/reduzida
  - à implementação dos dois operadores indicados!

Exemplo de algoritmo (pouco eficiente)

- Guardar todos os pares que vão sendo lidos
  - para cada novo par fazer uma pesquisa para descobrir se já estão ligados ou não
  - ➨ se o número de pares for muito grande a memória necessária pode ser demasiada
  - ➨ mesmo que guardassemos toda a informação já lida, como inferir se um dado par já está ligado?
    - Podiamos "aprender" novas ligações e criar novos pares
      ex: 2 4
      4 9 -> criar par 2 9
    - problema de memória piora ao aumentar o número de pares que guardamos
    - também aumenta o tempo de pesquisa para cada novo par

Algoritmo de procura rápida (Quick Find) [1]

- Dado ***(p, q),*** a dificuldade em saber se ***p*** e ***q*** já estão ligados não devia depender do número de pares anteriormente lido
  - estrutura de dados não pode crescer com cada par lido
- Se ***p*** e ***q*** estão ligados basta guardar um dos valores
  - por exemplo se para cada ***p*** guardarmos um único valor ***id[p]***, então se ***p*** for ligado a ***q*** os valores ***id[p]*** e ***id[q]*** podem ser iguais

  ➨ estrutura de dados ocupa memória fixa
    - eficiente em termos de espaço

  ➨ se a estrutura de dados é fixa o tempo necessário para a percorrer pode também ser fixo

Algoritmo de procura rápida (Quick Find) [2]

- Estrutura de dados a usar é uma tabela, ***id[]***
  - tamanho igual ao número de elementos possíveis
    - que admitimos ser conhecido à partida
- para implementar a função *procura*
  - apenas fazemos um teste para comparar ***id[p]*** e ***id[q]***
- para implementar a função de união
  - dado ***(p, q)*** percorremos a tabela e trocamos todas as entradas com valor ***p*** para terem valor ***q***
  - podia ser ao contrário (Exercício: verifique que assim é)

Pergunta: Se houver ***N*** objectos e viermos a fazer ***M*** uniões, quantas "instruções" (*procura*+*união*) é que são executadas?

Algoritmo de procura rápida (Quick Find) [3]

```
#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", &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);
   }
}
```

Algoritmo de procura rápida (Quick Find) [4]

- E se *N* não for fixo para todos os problemas?
- Que alteração são necessárias?

```
main (int argc, char *argv[])
{
  int i, p, q, t, N = atoi (argv[1]);
  int *id = (int *) malloc (N * sizeof (int));

  for (i = 0; i < N ; i ++)  id[i] = i ;

  while (scanf("%d %d", &p, &q) == 2) {
     if (id[p] == id[q])  continue;
     t = id[p];
     for (i = 0; i < N; i ++)
       if (id[i] == t) id[i] = id[q];
     printf (" %d %d\n", p, q);
  }
}
```

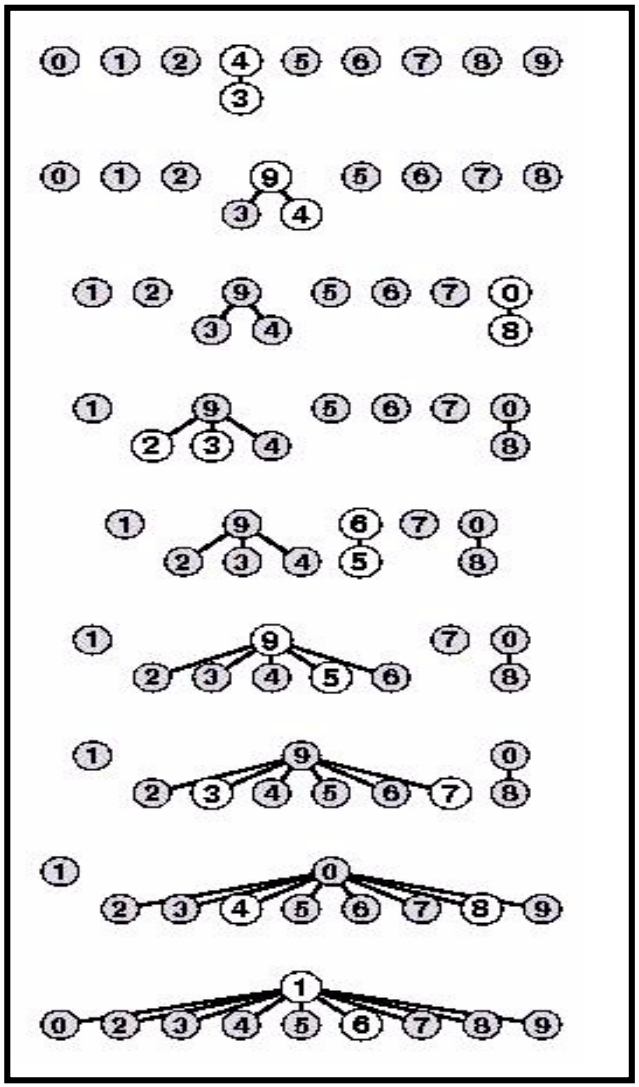
Algoritmo de procura rápida (Quick Find) - Exemplo de aplicação

Sequência ***p*** e ***q*** e tabela de conectividade

p q	**0 1 2 3 4 5 6 7 8 9**
3 4	0 1 2 4 4 5 6 7 8 9
4 9	0 1 2 9 9 5 6 7 8 9
8 0	0 1 2 9 9 5 6 7 0 9
2 3	0 1 9 9 9 5 6 7 0 9
5 6	0 1 9 9 9 6 6 7 0 9
2 9	0 1 9 9 9 6 6 7 0 9
5 9	0 1 9 9 9 9 9 7 0 9
7 3	0 1 9 9 9 9 9 9 0 9
4 8	0 1 0 0 0 0 0 0 0 0
0 2	0 1 0 0 0 0 0 0 0 0
6 1	1 1 1 1 1 1 1 1 1 1

Algoritmo de procura rápida (Quick Find) - Representação gráfica

- o nó ***p*** aponta para ***id[p]***
- nó superior representa componente



Análise - Algoritmo de procura rápida [1]

- Tempo necessário para efectuar *procura* é constante
  - **um** só teste
- A *união* é lenta
  - é preciso mudar muito entradas na tabela para cada união
  - demasiado lento se **N** for elevado
- Se ***p*** e ***q*** estão ligados, ou seja

  id[p] = id[q]

  e na entrada temos ***(q,r)*** porquê percorrer toda a tabela para mudar ***id[p] = id[q] = r***?
  - seria melhor apenas anotar que todos os elementos anteriormente ligados a ***q*** agora estavam ligados também a ***r***

Análise - Algoritmo de procura rápida [2]

- Máquinas existentes em 2000:
  - **10^9** operações por segundo
  - **10^9** palavras de memória
    - se acesso a cada palavra aproximadamente 1s
- Procura rápida com **10^{10}** pares de dados e **10^9** ligações
    - não são exagerados para uma rede telefónica ou um circuito integrado
  - requer **10^{20}** operações **~ 3000 anos de computação**
- Se número de nós ***O(N)*** e número de ligação ***O(N)***
  - tempo de execução $\propto$ ***$O(N^2)$*** => complexidade!
- Se usarmos um computador 10 vezes mais rápido (e com 10 vezes mais memória)
  - demora 10 vezes mais a processar 10 vezes mais dados!!

    Exercicio: demonstre que assim é!

Algoritmo de união rápida (Quick Union) [1]

- Estrutura de dados é a mesma, mas interpretação é diferente
  - inicialização é a mesma (***id[p] = p***, $\forall p = 1,...,N$)
  - cada objecto aponta para outro objecto no mesmo conjunto
  - estrutura não tem ciclos a efectuar procura:
- Partindo de cada objecto seguimos os objectos que são apontados até um que aponte para ele próprio
  - dois elementos estão no mesmo conjunto se este processo levar em ambos os casos ao mesmo elemento
- Se não estão no mesmo conjunto colocamos um a apontar para o outro
  - operação extremamente rápida e simples

Algoritmo de união rápida (Quick Union) [2]

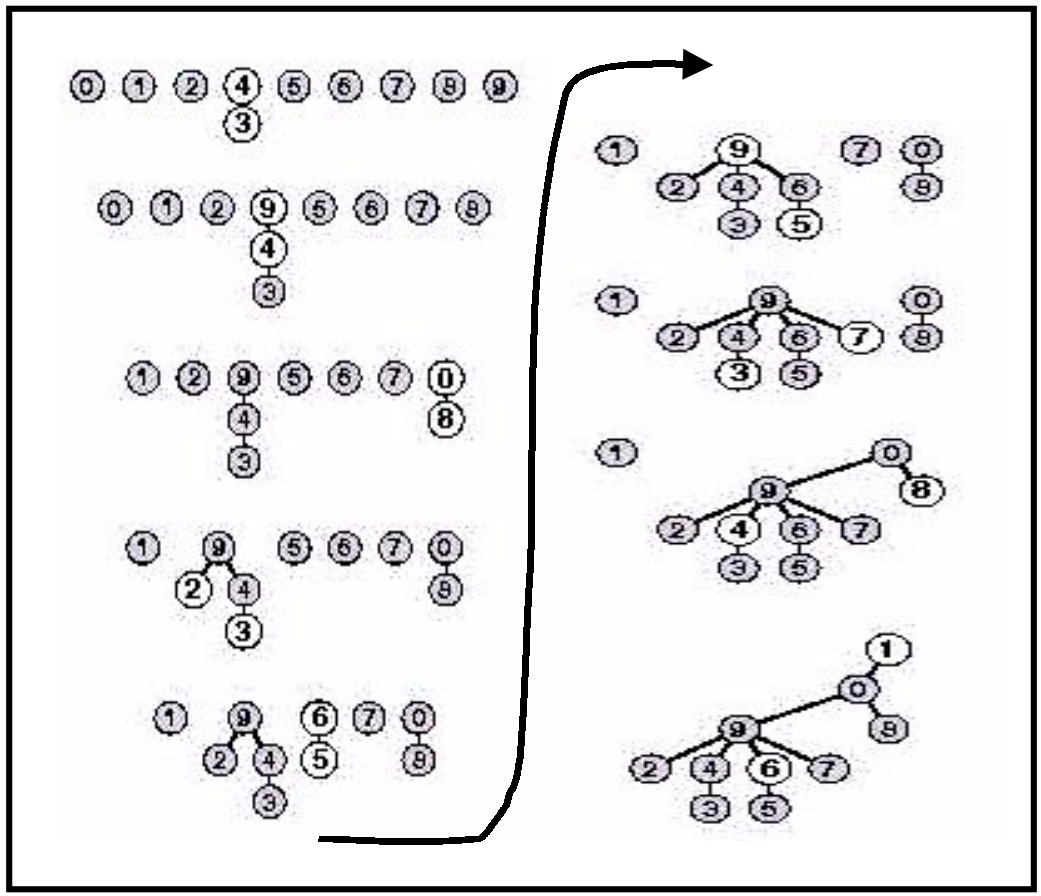
```
main (int argc, char *argv[])
{
  int i, j, p, q, t, N = atoi (argv[1]);
  int *id = (int *) malloc (N * sizeof (int));

  for (i = 0; i < N ; i ++) id[i] = i ;

  while (scanf("%d %d", &p, &q) == 2) {
     i = p; j = q;
     while (i != id[i]) i = id[i];
     while (j != id[j]) j = id[j];
     if (i == j)  continue;
     id[i] = j;
     printf (" %d %d\n", p, q);
  }
}
```

Algoritmo de união rápida (Quick Union) - Exemplo de aplicação

p q	**0 1 2 3 4 5 6 7 8 9**
3 4	0 1 2 4 4 5 6 7 8 9
4 9	0 1 2 4 9 5 6 7 8 9
8 0	0 1 2 4 9 5 6 7 0 9
2 3	0 1 9 4 9 5 6 7 0 9
5 6	0 1 9 4 9 6 6 7 0 9
5 9	0 1 9 4 9 6 9 7 0 9
7 3	0 1 9 4 9 6 9 9 0 9
4 8	0 1 9 4 9 6 9 9 0 0
6 1	1 1 9 4 9 6 9 9 0 0

Algoritmo de união rápida (Quick Union) - Representação gráfica

- *procura* percorre caminho de forma ascendente
  - ver se dois caminhos chegam ao mesmo nó)
- *união* liga vários caminhos



Algoritmo de união rápida (Quick Union) - Propriedades

- Estrutura de dados é uma árvore
  - estudaremos mais detalhadamente mais adiante
- *união* é muito rápida mas a *procura* é mais lenta
  - algoritmo é de facto mais rápido?
  - díficil de demonstrar
    - por análise detalhada é possível demonstrar que é mais rápido se os dados forem aleatórios
    - "caminho" vertical pode crescer indefinidamente

Algoritmo de união rápida (Quick Union) - Caso patológico

- Suponha que os dados de entrada estão na sequência natural

  1 2
  2 3
  3 4
  ...

- A representação gráfica é apenas uma linha vertical com *N* elementos

  N aponta para N-1
  N-1 aponta para N-2
  …
  2 aponta para 1

- *procura* pode demorar aproximadamente ***N*** passos
- Para **N** dados custo total cresce com N^2

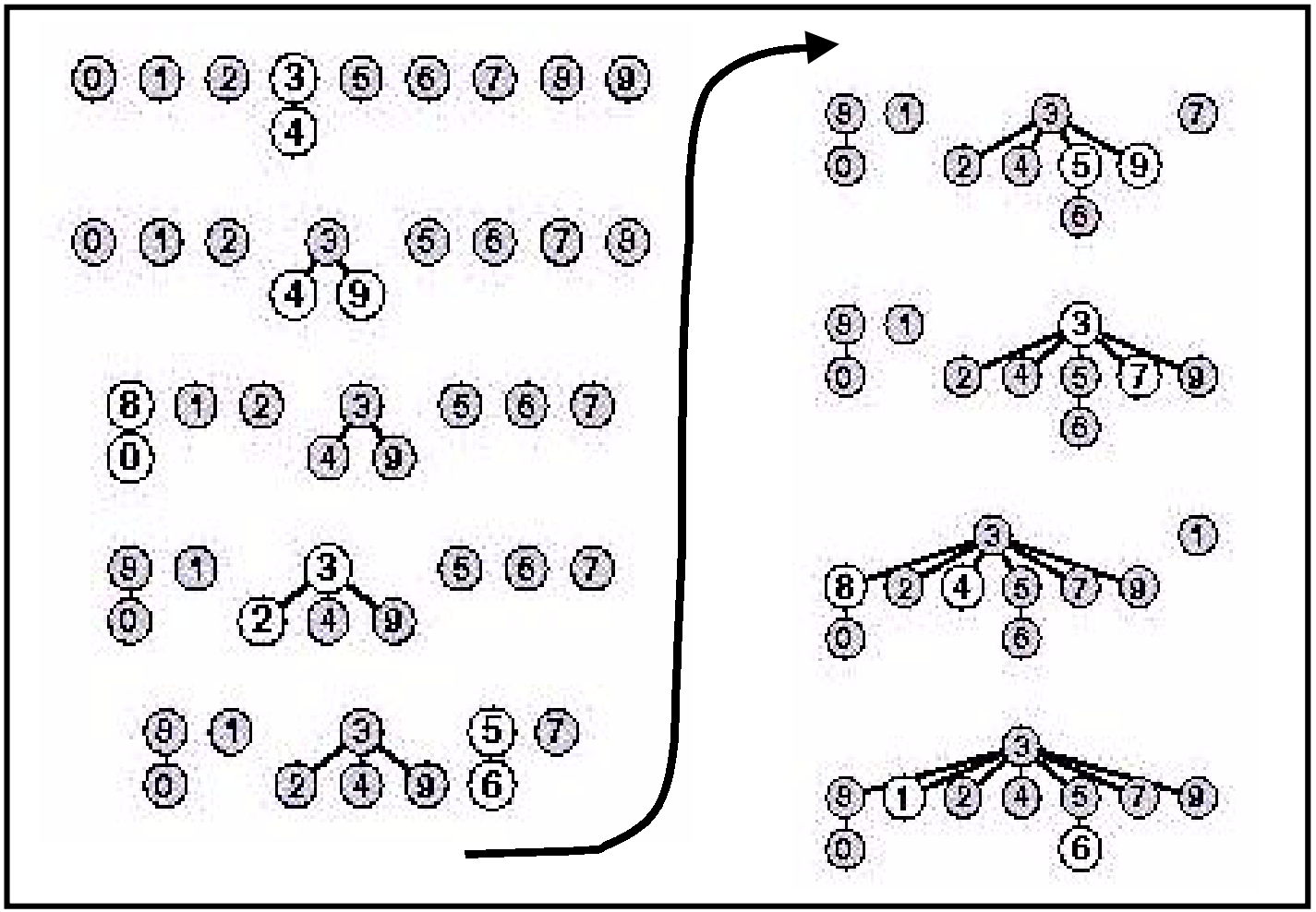
Algoritmo de união rápida equilibrada (Weighted Union) [1]

- Evitar crescimento demasiado rápido
  - manter informação sobre o estado de cada componente
  - equilibrar ligando componentes pequenos debaixo de outros maiores
- Ideia chave:quando se efectua a união de dois conjuntos, em vez de ligar a árvore de um ao outro, ligamos sempre a árvore mais pequena à maior
  - necessário saber o número de nós em cada sub-árvore
    - Requer mais estruturas de dados
- Mais código para decidir/descobrir onde efectuar a ligação

Algoritmo de união rápida equilibrada (Weighted Union) [3]

```
main(int argc, char *argv[])
{
   int i, j, p, q, t, N = atoi(argv[1]);
   int *id = (int *) malloc(N * sizeof(int));
   int *sz = (int *) malloc(N * sizeof(int));

   for (i = 0; i < N; i++) {
      id[i] = i; sz[i] = 1;
   }

   while (scanf("%d %d", &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);
   }
}
```

Algoritmo de união rápida equilibrada (Weighted Union) - Representação Gráfica



Algoritmo de união rápida equilibrada (Weighted Union) - Pior Caso [1]

p q	**0 1 2 3 4 5 6 7 8 9**
0 1	0 0 2 3 4 5 6 7 8 9
2 3	0 0 2 2 4 5 6 7 8 9
4 5	0 0 2 2 4 4 6 7 8 9
6 7	0 0 2 2 4 4 6 6 8 9
8 9	0 0 2 2 4 4 6 6 8 8
0 2	0 0 0 2 4 4 6 6 8 8
4 6	0 0 0 2 4 4 4 6 8 8
0 4	0 0 0 2 0 4 4 6 8 8
6 8	0 0 0 2 0 4 4 6 0 8

Algoritmo de união rápida equilibrada (Weighted Union) - Pior Caso [2]

- A altura da maior árvore apenas aumenta quando é ligada a uma árvore igualmente "grande"
- Ao juntar duas árvores com 2^N nós obtemos uma árvore com $2^{(N+1)}$ nós
  - máxima distância para a raiz aumenta de N para $N+1$ (no máximo)
- O custo do algoritmo para determinar se dois elementos estão ligados é da ordem de $lg\ N$
  - para 10^{10} pares de dados e 10^9 ligações

    tempo é reduzido de **3000** anos para **1** minuto!!

Exemplo de conectividade - Melhorar o desempenho

- Análise empírica demonstra que o algoritmo de união rápida equilibrada pode típicamente resolver problemas prácticos em tempo linear
  - custo de execução do algoritmo está apenas a um factor constante do custo (**inevitável**) de leitura dos dados
- Garantir que o tempo de execução é de facto linear é difícil
- É possível ainda melhorar o desempenho do algoritmo através de uma técnica de compressão
  - fazer todos os nós visitados apontar para a raíz
  - denomina-se algoritmo de caminho comprimido (ver livro!)

Algoritmo de união rápida equilibrada com compressão de caminho [1]

```
main(int argc, char *argv[])
{
   int i, j, p, q, t, N = atoi(argv[1]);
   int *id = (int *) malloc(N * sizeof(int));
   int *sz = (int *) malloc(N * sizeof(int));

   for (i = 0; i < N; i++) {
      id[i] = i; sz[i] = 1;
   }

   while (scanf("%d %d", &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]; t = j;
      } else {
         id[j] = i; sz[i] += sz[j]; t = i;
      }
      for (i = p; i != id[i]; i = id[i]) id[i] = t;
      for (j = q; j != id[j]; j = id[j]) id[j] = t;
      printf(" %d %d\n", p, q);
   }
}
```

Algoritmo de união rápida equilibrada com compressão de caminho [2]

- No pior caso a altura da árvore é ***O(lg*N)***
  - demonstração é muito complicada
  - **mas** o algoritmo é muito simples!!
- Notar que ***lg*N*** é um valor muito pequeno
  - quase constante

N	***lg*N***
2	1
4	2
16	3
65536	4
qualquer valor realista	5

- O custo de execução do algoritmo está apenas a um factor constante do custo (inevitável) de leitura dos dados

Exemplo de conectividade - Comparação [1]

- Desempenho no pior caso:

N	M	F	U	W	P	H
1000	6206	14	25	6	5	3
2500	20236	82	210	13	15	12
5000	41913	304	1172	46	26	25
10000	83857	1216	4577	91	73	50
25000	309802			219	208	216
50000	708701			469	387	497
100000	1545119			1071	1106	1096

F - Quick Find
U - Quick Union
W - Weighted Quick Union
P - Weighted Quick Union com compressão
H - Weighted Quick-Union com divisão

Custo por ligação: (pior caso)

Quick Find - ***N***
Quick Union - ***N***
Weighted - ***lg N***
Path Compression - ***5***

Exemplo de conectividade - Comparação [2]

- Algoritmo de tempo-real: resolve o problema enquanto lê os dados
  - como o custo é constante ou da ordem de grandeza do custo de ler os dados, tempo de execução é negligenciável:
    - execução sem custos!!
- Estrutura de dados apropriada ao algoritmo
- Abstração de operações entre conjuntos:
  - *procura*: será que ***p*** está no mesmo conjunto que ***q***?
  - *união*: juntar os conjuntos em que ***p*** e ***q*** estão contidos
  - abstracção útil para muitas aplicações

Exemplo de conectividade - Aspectos Fundamentais

- Começar sempre por definir bem o problema
- Começar sempre com uma solução através de um algoritmo simples
- Não usar um algoritmo demasiado simples para problemas complicados
  - Impossível usar um algoritmo demasiado simples para problemas de grande dimensão
- A abstracção ajuda na resolução do problema
  - importante identificar a abstracção fundamental
- O desempenho rápido em dados realistas é bom, mas
  - procure obter garantias de desempenho em casos patológicos

Análise de Algoritmos [1]

- Para avaliar e comparar o desempenho de dois algoritmos:
  - executamos ambos (muitas vezes) para ver qual é mais rápido
  - fornece indicações sobre o desempenho e informação sobre como efectuar uma análise mais profunda
- Método empírico pode consumir demasiado tempo
  - é necessário uma análise mais detalhada para validar resultados
- Que dados usar?
  - **dados reais**: verdadeira medida do custo de execução
  - **dados aleatórios**: assegura-nos que as experiências testam o algoritmo e não apenas os dados específicos
    - Caso médio
  - **dados perversos**: mostram que o algoritmo funciona com qualquer tipo de dados
    - Pior caso!
  - **dados benéficos**:
    - Melhor caso

Análise de Algoritmos [2]

- Melhorar algoritmos
  - analisando o seu desempenho
  - fazendo pequenas alterações para produzir um novo algoritmo
  - identificar as abstracções essenciais do problema
  - comparar algoritmos com base no seu uso dessas abstracções
- Por vezes ignoram-se as características essenciais de desempenho
  - aceita-se um algoritmo mais lento para evitar analisá-lo ou fazer alterações complicadas
  - é no entanto possível obter tremendos aumentos de desempenho escolhendo/desenvolvendo o algoritmo certo
- Evitar perder demasiado tempo a estudar essas características
  - para quê optimizar o desempenho de um algoritmo que
    - já é rápido no contexto da sua utilização
    - é utilizado com poucos dados
    - ou é pouco utilizado (executado)

Análise de Algoritmos [3]

- É fundamental para percebermos um algoritmo de forma a tomarmos partido dele de forma efectiva/eficiente
  - compararmos com outros
  - prevermos o desempenho
  - escolher correctamente os seus parâmetros
- Importante saber que há bases científicas irredutíveis para caracterizar, descrever e comparar algoritmos

Intenção nesta disciplina é

- ilustrar o processo
- introduzir alguma notação
- introduzir este tipo de análise demonstrando a sua utilidade e importância

Análise de Algoritmos [4]

- Análise precisa é uma tarefa complicada:
  - algoritmo é implementado numa dada linguagem
  - linguagem é compilada e programa é executado num dado computador
  - difícil prever tempos de execução de cada instruções e antever optimizações
  - muitos algoritmos são "sensíveis" aos dados de entrada
  - muitos algoritmos não são bem compreendidos
- É em geral no entanto possível prever precisamente o tempo de execução de um programa,
  - ou que um algoritmo é melhor que outro em dadas circunstâncias (bem definidas)
    - apenas é necessário um pequeno conjunto de ferramentas matemáticas

Análise de Algoritmos [5]

- Fundamental é separar a análise da implementação, i.e. identificar as operações de forma abstracta
  - é relevante saber quantas vezes ***id[p]*** é acedido
    - Uma propriedade (**imutável**) do algoritmo
  - não é tão importante saber quantos nanosegundos essa instrução demora no meu computador!
    - Uma propriedade do computador e **não** do algoritmo
- O número de operações abstractas pode ser grande, mas
  - o desempenho tipicamente depende apenas de um pequeno número de parâmetros
  - procurar determinar a frequência de execução de cada um desses operadores
    - estabelecer estimativas

Análise de Algoritmos [6]

- Depêndencia nos dados de entrada
  - dados reais geralmente não disponíveis:
  - assumir que são aleatórios: **caso médio**
    - por vezes a análise é complicada
    - Pode ser uma ficção matemática não representativa da vida real
  - perversos: **pior caso**
    - por vezes difícil de determinar
    - podem nunca acontecer na vida real
  - benéficos: **melhor caso**

- Em geral são boas indicações quanto ao desempenho de um algoritmo

Análise: Crescimento de Funções [1]

- O tempo de execução geralmente dependente de um único parâmetro ***N***
  - ordem de um polinómio
  - tamanho de um ficheiro a ser processado, ordenado, etc
  - ou medida abstracta do tamanho do problema a considerar
  - ➨ usualmente relacionado com o número de dados a processar

- Quando há mais de um parâmetro
  - procura-se exprimir todos os parâmetros em função de um só
  - faz-se uma análise em separado para cada parâmetro

Análise: Crescimento de Funções [2]

- Os Algoritmos têm tempo de execução proporcional a:
  - ***1*** - muitas instruções são executadas uma só vez ou poucas vezes
    - se isto for verdade para todo o programa diz-se que o seu tempo de execução é constante
  - ***log N*** - tempo de execução é logarítmico
    - cresce ligeiramente à medida que ***N*** cresce
    - usual em algoritmos que resolvem um grande problema reduzindo-o a problemas menores e resolvendo estes
    - quando ***N*** duplica ***log N*** aumenta mas muito pouco; apenas duplica quando ***N*** aumenta para N^2
  - ***N*** - tempo de execução é linear
    - típico quando algum processamento é feito para cada dado de entrada
    - situação óptima quando é necessário processar ***N*** dados de entrada (ou produzir ***N*** dados na saída)

Análise: Crescimento de Funções [3]

N log N - típico quando se reduz um problema em sub-problemas, se resolve estes separadamente e se combinam as soluções

- se ***N*** é 1 milhão ***N log N*** é perto de 20 milhões

N^2 - tempo de execução quadrático

- típico quando é preciso processar todos os pares de dados de entrada
- prático apenas em pequenos problemas (ex: produto matriz - vector)

N^3 - tempo de execução cúbico

- para $N = 100$, $N^3 = 1$ milhão (ex: produto de matrizes)

2^N - tempo de execução exponencial

- provavelmente de pouca aplicação prática
- típico em soluções de força bruta
- para $N = 20$, $2^N = 1$ milhão; N duplica, tempo passa a ser o quadrado (ex: cálculo da saída de um circuito lógico de N entradas)

Análise: Crescimento de Funções [4]

- outros exemplos são

log log N

log* N (número de logs até 1)

➔ Usualmente muito mais pequenas (quase constantes)

Valores típicos de várias funções

$lg\ N$	$\sqrt{N}$	N	$N\ lg\ N$	$N\ (lg\ N)^2$	$N^{3/2}$	N^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	1000000	100000000
17	316	100000	1660964	27588016	31622777	10000000000
20	1000	1000000	19931569	397267426	1000000000	1000000000000

Análise: Resolução de grandes problemas [1]

- Relação complexidade/tempo de execução
  - conversão para unidades de tempo

segundos	
10^2	1.7 minutos
10^4	2.8 horas
10^5	1.1 dias
10^6	1.6 semanas
10^7	3.8 meses
10^8	3.1 anos
10^9	3.1 décadas
10^{10}	3.1 séculos
10^{11}	*nunca*

Análise: Resolução de grandes problemas [2]

operações por segundo	tamanho do problema - 1 milhão				tamanho do problema - 1 billião		
	N	***N lg N***	N^2		***N***	***N lg N***	N^2
10^6	segundos	segundos	semanas		horas	horas	*nunca*
10^9	*instantâneo*	*instantâneo*	horas		segundos	segundos	décadas
10^{12}	*instantâneo*	*instantâneo*	segundos		*instantâneo*	*instantâneo*	semanas

Análise: Complexidade

- Complexidade refere-se ao tempo de execução
  - usualmente o termo de maior ordem domina (para **N** elevado)
  - comum todos os termos serem multiplicados por uma constante
  - para ***N*** pequeno vários termos podem ser igualmente relevantes
    - ex: $\mathbf{0.1}\ N^3 + \mathbf{100} * N^2$ para $\mathbf{1} \leq N \leq \mathbf{1000}$
  - base do logaritmo não é muito relevante
    - Mudar de base é um factor constante
    - Bases mais comuns são **2** (***lg N***), **10** ***(log N)*** e **e** (***ln N)***
      - [e = número de Neper]
- Análise de complexidade é muito importante quando ***N*** aumenta

Análise: Funções Relevantes

função	*nome*	*valores típicos*	*aproximação*
$\lfloor x \rfloor$	função floor (chão)	$\lfloor 3.14 \rfloor = 3$	x
$\lceil x \rceil$	função ceiling (tecto)	$\lceil 3.14 \rceil = 4$	x
$\lg N$	logaritmo binário	$\lg 1024 = 10$	$1.44 \ln N$
F_N	números de Fibonacci	$F_{10} = 55$	$\phi^N / \sqrt{5}$
H_N	números harmónicos	$H_{10} \approx 2.9$	$\ln N + \gamma$
$N!$	função factorial	$10! = 3628800$	$(N/e)^N$
$\lg (N!)$		$\lg(100!) \approx 520$	$N \lg N - 1.44\ N$

$e = 2.71828 \ldots$

$\gamma = 0.57721 \ldots$

$\phi = (1 + \sqrt{5}) / 2 = 1.61803 \ldots$

$\ln 2 = 0.693147 \ldots$

$\lg e = 1 / \ln 2 = 1.44269 \ldots$

Análise: Sucessões

- ***log N*** - área debaixo da curva de ***1/x*** entre ***1*** e ***N*** (integração)

  $H_N \approx \ln N + \gamma + 1/(12N)$ $\quad\quad$ $\gamma = 0.57721$ (constante de Euler)

  é uma boa aproximação $\quad\quad$ reflecte diferenca entre H_N e o integral

- Números de Fibonacci

  $F = F_{N-1} + F_{N-2}, \quad para \quad N \geq 2 \quad com \quad F_0 = 0 \quad e \quad F_1 = 1$

- Fórmula de Stirling

  $\lg N! \approx N \lg N - N \lg e + \lg \sqrt{2\pi N}$

- e muitas outras:
  - distribuição binomial
  - aproximação de Poisson
  - etc

Análise: Notação "O grande" [1]

- A notação matemática que nos permite suprimir detalhes na análise de algoritmos

- **<u>Definição</u>:** uma função ***g(N)*** diz-se ser ***O(f(N))*** se existem constantes c_0 e N_0 tais que $g(N) < c_0 f(N)$ para qualquer $N > N_0$

- A notação é usada com três objectivos:
  - limitar o erro que é feito ao ignorar os termos menores nas fórmulas matemáticas
  - limitar o erro que é feito na análise ao desprezar parte de um programa que contribui de forma mínima para o custo/complexidade total
  - permitir-nos classificar algoritmos de acordo com limites superiores no seu tempo de execução

Análise: Notação "O grande" [2]

- Os Resultados da análise não são exactos
  - são aproximações tecnicamente bem caracterizadas
- Manipulações com a notação "O grande"
  - permitem-nos ignorar termos de menor importância de forma precisa
  - podem ser feitas como se o "O" não estivesse lá

Ex: $(N + O(1))\ (N + O(\log N)) =$

usando distributividade e o facto de que

$f(N) \cdot O(g(N)) = O(f(N) \cdot g(N))$ se $f(N)$ não for constante

$O(1) \cdot g(N) = O(g(N))$ fica

$= N^2 + O(N) + O(N \log N) + O(\log N) =$

e como $O(N \log N) > O(N) > O(\log N)$

$= N^2 + O(N \log N)$

Análise: Notação "O grande" [3]

- Manipulações com a notação "O grande": alguns exemplos
  - $O(f) + O(g) = O(f+g) = O(max(f,g))$
  - $O(f) \,.\, O(g) = O(f \,.\, g)$
  - $O(f) + O(g) = O(f)$ sse $g(N) \leq f(N)$ para $\forall\, N > N_0$
  - $O(c\, f(N)) = O(f(N))$
  - $f(N) = O(f(N))$

Fórmula com termo contendo ***O(...)*** diz-se

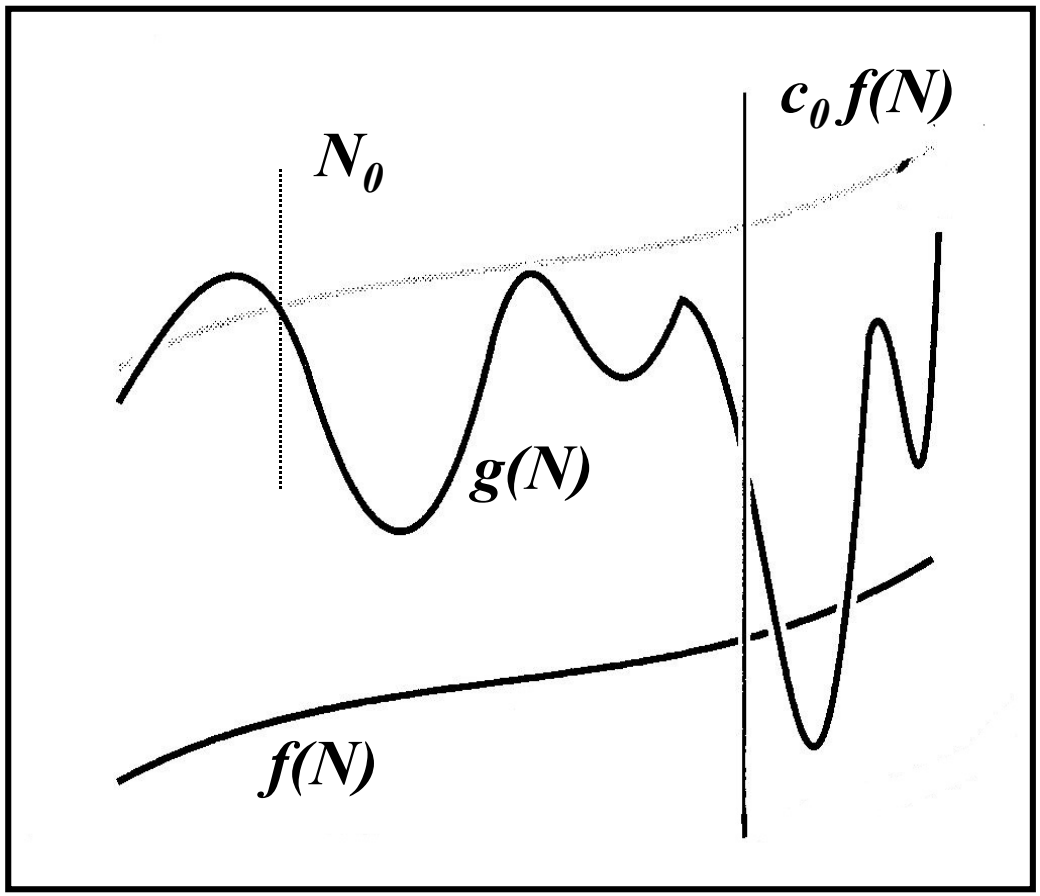
<u>expressão assimptótica</u>

Notação O grande - Representação Gráfica

- Limitar uma função com uma aproximação O

  $g(n) = O(f(N))$

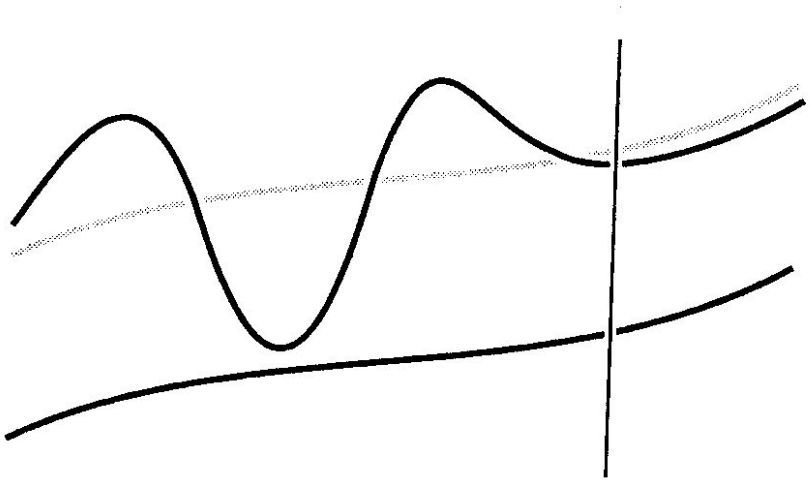
  - *g(N)* debaixo de uma curva com o andamento de *f()* a partir de um dado *N*

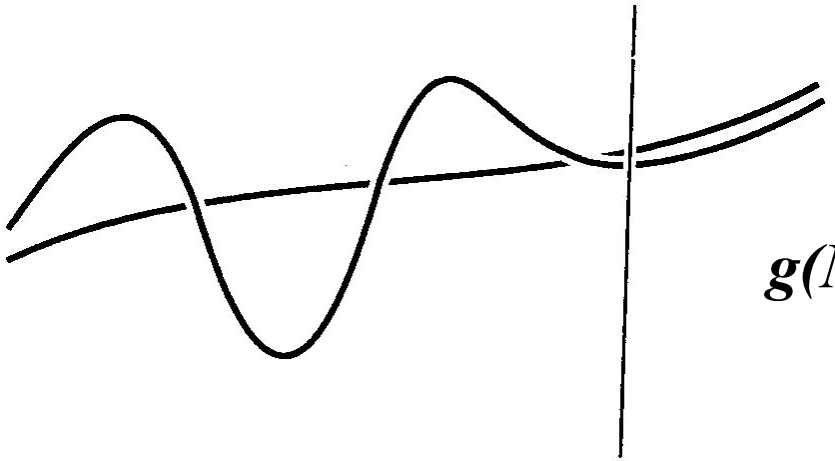


Análise: Aproximação funcional

g(N) proporcional a ***f(N)***

- ***g()*** eventualmente cresce como ***f()***, a menos de uma constante





g(N) "é como" ***f(N)***

- permite usar ***f()*** para estimar ***g()***

Análise: Recorrências básicas [1]

- Muitos algoritmos são baseados na ideia de "dividir para conquistar”
  - dividir recursivamente um problema grande em vários problemas menores
  - recursividade é uma técnica muito efectiva (estudaremos posteriormente)
- É importante perceber como analizar algumas fórmulas standard
  - intervêm muitas vezes na análise de muitos algoritmos
  - permite compreender os métodos básicos de análise

Análise: Recorrências básicas [2]

Programa recursivo que analisa todos os dados de entrada em cada passo para eliminar um item

$C_N = C_{N-1} + N$ para $N \geq 2$ com $C_1 = 1$

Solução: C_N "é" aproximadamente $N^2/2$

Demonstração: usar a propriedade telescópica e aplicar a fórmula a si própria ■

$$
\begin{aligned}
C_N &= C_{N-1} + N \\
&= C_{N-2} + (N-1) + N \\
&= C_{N-3} + (N-2) + (N-1) + N \\
&\vdots \\
&= C_1 + 2 + \ldots + (N-2) + (N-1) + N \\
&= 1 + 2 + \ldots + (N-2) + (N-1) + N \\
&= \frac{N(N+1)}{2}
\end{aligned}
$$

Importante: O carácter recursivo de um algoritmo reflecte-se directamente na sua análise

Análise: Recorrências básicas [3]

Programa recursivo que, em cada passo, divide em dois os dados de entrada em cada passo

$C_N = C_{N/2} + 1$ para $N \geq 2$ com $C_1 = 1$

Solução: C_N é aproximadamente $\log N$

Demonstração: $N=2^n$ ($n = \log N$);
usar a propriedade telescópica e aplicar a fórmula a si própria ■

$$\begin{aligned} C_{2^n} &= C_{2^{n-1}} + 1 \\ &= C_{2^{n-2}} + 1 + 1 \\ &= C_{2^{n-3}} + 3 \\ &\vdots \\ &= C_{2^0} + n \\ &= n + 1 \end{aligned}$$

Interpretação: se $N/2$ for $\lfloor N/2 \rfloor$, então C_N é o número de bits na representação binária de N, logo é $\lfloor \lg N \rfloor + 1$

Análise: Recorrências básicas [4]

Programa recursivo que, em cada passo, divide em duas metades os dados de entrada, mas tem de examinar todos os items

$$C_N = C_{N/2} + N \quad \text{para} \quad N \geq 2 \quad \text{com} \quad C_1 = 0$$

Solução: C_N é aproximadamente $2N$

Demonstração: usar a propriedade telescópica e aplicar a fórmula a si própria

A recorrência leva à soma $N + N/2 + N/4 + N/8 + \ldots$

Se a sequência for infinita, a soma da série geométrica é $2N$ ■

Análise: Recorrências básicas [5]

Programa recursivo que tem de examinar todos os dados de entrada antes, durante ou depois de os dividir em duas metades

$$C_N = 2\,C_{N/2} + N \quad \text{para} \quad N \geq 2 \text{ com } C_1 = 0$$

Solução: C_N é aproximadamente $N \lg N$

Demonstração: $N=2^n$ ($n = \log N$); (como indicado) ■

$$\begin{aligned} C_{2^n} &= 2\,C_{2^{n-1}} + 2^n \\ \frac{C_{2^n}}{2^n} &= \frac{C_{2^{n-1}}}{2^{n-1}} + 1 \\ &= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1 \\ &\vdots \\ &= n \end{aligned}$$

Análise: Recorrências básicas [6]

Programa recursivo que divide os dados de entrada em cada passo mas um número constante de outras operações

$$C_N = 2\,C_{N/2} + 1 \quad \text{para} \quad N \geq 2 \quad \text{com} \quad C_1 = 1$$

Solução: C_N é aproximadamente $2N$

Demonstração: (como a anterior) ■

Análise: Procura sequencial e binária

- Dois algoritmos básicos que nos permitem ver se um elemento de uma sequência de objectos aparece num conjunto de objectos previamente guardados
  - permite ilustrar o processo de análise e comparação de algoritmos
- Exemplo de aplicação:
  - companhia de cartões de crédito quer saber se as últimas ***M*** transacções envolveram algum dos ***N*** cartões dados como desaparecidos ou de maus pagadores
- Pretende-se estimar o tempo de execução dos algoritmos
  - ***N*** é muito grande (ex: 10^4 a 10^6) e ***M*** enormíssimo (10^6 a 10^9)

Procura Sequencial - Solução trivial

Assumir que

- os objectos são representados
  por números inteiros
- dados armazenados numa tabela
- procura é sequencial na tabela

```
int search (int a[], int v, int l, int r)
{
   int i;
   for (i = l; i <= r; i++)
      if (v == a[i]) return i ;
   return -1;
}
```

Procura Sequencial - Solução trivial Análise [1]

- O tempo de execução depende se o objecto está ou não na tabela
  - se não estiver percorremos a tabela toda (***N*** elementos)
  - se estiver podemos descobrir logo no início (ou no fim)
  - tempo depende dos dados!
- Assumir algo sobre os dados:
  - números são aleatórios (não relacionados entre si) ??
  - esta propriedade é critica!
- Assumir algo sobre a procura
  - estudar o caso de sucesso e o de insucesso separadamento
  - tempo de comparação assumido constante

Procura Sequencial - Solução trivial Análise [2]

Propriedade: na procura sequencial o número de elementos da tabela que são examinados é:

- ***N*** em caso de insucesso
- em média aproximadamente ***N/2*** em caso de sucesso

– Tempo de execução é portanto proporcional a ***N*** (linear)!

– Isto para **cada** elemento de entrada

- Custo total é ***O(MN)*** que é enorme

- Como melhorar o algoritmo?
  – manter os números ordenados na tabela (estudaremos algoritmos de ordenação nos capítulos seguintes)
    - na procura sequencial numa tabela ordenada o custo é ***N*** no pior caso e ***N/2*** em média (quer haja sucesso ou não)

Procura Binária [1]

- Se demorar ***c*** microsegundos a examinar um número e ***M***=10^9, ***N***=10^6
  - tempo total de verificação são 16 ***c*** anos !!

- **Ideia**: se os números na tabela estão ordenados podemos eliminar metade deles comparando o que procuramos com o que está na posição do meio
  - se for igual temos sucesso (sorte!)
  - se for menor aplicamos o mesmo método à primeira metade da tabela
  - se for maior aplicamos o mesmo método à segunda metade da tabela

Procura Binária [2]

```
int search (int a[], int v, int l, int r)
{
   while (r >= l) {
      int m = (l+r) / 2 ;

      if (v == a[m]) return m ;
      if (v < a[m]) r = m-1;
      else l = m+1 ;
   }
   return -1 ;
}
```

Procura Binária - Análise

Propriedade: A procura binária nunca examina

mais do que $\lfloor lg\ N \rfloor + 1$ números

Demonstração: (ilustra uso de recorrências)

Seja T_N o número de comparações no pior caso; redução em 2 implica $T_N \leq T_{\lfloor N/2 \rfloor} + 1$ para $N \geq 2$ com $T_1 = 1$, i.e., após uma comparação ou temos sucesso ou continuamos a procurar numa tabela com $\lfloor N/2 \rfloor$ elementos

Sabemos de imediato que $T_N \leq n+1$ (em que $N = 2^n$) e o resto segue por indução matemática) ■

- Mostra que este tipo de pesquisa permite resolver problemas até um milhão de dados com cerca de 20 comparações por procura
  - possivelmente menos do que o tempo que demora a ler ou escrever os números num computador real

Estudo empírico de algoritmos de procura

	M = 1000		*M = 10000*		*M = 100000*	
N	***S***	***B***	***S***	***B***	***S***	***B***
125	***1***	***1***	***13***	***2***	***130***	***20***
250	***3***	***0***	***25***	***2***	***251***	***22***
500	***5***	***0***	***49***	***3***	***492***	***23***
1250	***13***	***0***	***128***	***3***	***1276***	***25***
2500	***26***	***1***	***267***	***3***		***28***
5000	***53***	***0***	***533***	***3***		***30***
12500	***134***	***1***	***1337***	***3***		***33***
25000	***268***	***1***		***3***		***35***
50000	***537***	***0***		***4***		***39***
100000	***1269***	***1***		***5***		***47***

S - Procura Sequencial B - Procura binária

Conclusões da análise do exemplo

- Identificação de operações básicas de forma abstracta
- Utilização de análise matemática para estudar a frequência com que um algoritmo executa essas operações
- Utilizar resultados para deduzir uma estimativa funcional do tempo de execução
- Verificar e estender estudos empíricos
- Identificar e eventualmente optimizar as características mais relevantes de um dado algoritmo

Garantias, Previsões e Limitações [1]

- O tempo de execução dos algoritmos depende criticamente dos dados.
- Objectivo da análise é
  - eliminar essa dependência
  - inferir o máximo de informação assumindo o mínimo possível
- O estudo do pior caso é importante:
    - permite obter **garantias** máximas
  - se o resultado no pior caso é aceitável, então a situação é favorável
    - é desejável utilizar algoritmos com bom desempenho no pior caso
  - se o resultado no pior caso for mau, pode ser problemático
    - não sabemos quão provável será aparecerem dados que levem a esse desempenho
  - importante não sobrevalorizar obtenção de valores baixos para o pior caso
    - não é relevante ter um bom algoritmo para o pior caso que é mau para dados típicos

Garantias, Previsões e Limitações [2]

- O estudo do desempenho no caso médio é atractivo:
  - Permite fazer **previsões**
  - bom quando é possível caracterizar muito bem os dados de entrada (tipo, frequência, etc)
  - mau quando tal não é possível/fácil

    ex: processar texto aleatório num ficheiro versus processar o texto de um livro de Eça de Queiróz!
  - análise pode ser não trivial em termos matemáticos

    ex: no algoritmo de união rápida
  - saber o valor médio pode não ser suficiente
    - podemos ter de saber o desvio padrão ou outras características
  - não permite dizer nada sobre a possíbilidade de um algoritmo ser dramaticamente mais lento que o caso médio
  - por vezes aleatoriedade pode ser imposta

Garantias, Previsões e Limitações [3]

- Análise descrita permite obter limites superiores ou valores esperados (em certos casos)
  - por vezes é relevante obter também limites inferiores!
- Se os limites inferiores e superiores forem próximos
  - é infrutífero tentar optimizar mais um algoritmo ou processo
  - mais vale tentar optimizar a implementação
- Análise pode dar indicações preciosas sobre como e onde melhorar o desempenho de um algoritmo
  - nem sempre é o caso no entanto!