Explotación de la información

Práctica 1 - Tokenizador
Cálculo de la complejidad temporal y espacial

1. Cálculo de la complejidad temporal
   (a) Casos especiales
      i. URL
      ii. Email
      iii. Guion
      iv. Generico
      v. Acronimo
      vi. Números
   (b) Otros métodos usados en los casos especiales
2. Cálculo de la complejidad espacial al tokenizar los ficheros

1.A Casos especiales

```
Void Tokenizador::URL(char*& p_izq, char*& p_der) const {
        if (p_izq != p_der) {
                if (EsDelimiter(*p_der+1) || *(p_der+1) == '\0') {
                        ...
                } else if (EsURLIndicador(ObtenerString(p_izq, p_der))){
                        while (*p_der != '\0' && !EsURLDelimiter(p_der)) { ++p_der; }
                }
        } else {++p_der;}
}
Bool Tokenizador::EsURLDelimiter( char*& p_caracter) const {
        return (*p_caracter == ' ') ||
                (_delimiters.find(*p_caracter) != std::string::npos
                        && _URLdelimiters.find(*p_caracter) == std::string::npos);
}
bool Tokenizador::EsURLIndicador(const std::string& p_indicador) const {
        return (_URLindc.find(p_indicador) != std::string::npos);
}
```

N = caracteres que forman el subtring que está tratando
L = longitud del string a buscar el carácter

→ **EsURLDelimiter() y EsURLIndicador**
O(L) => L + 1 // La documentación find(), tiene complejidad lineal

→ **URL()**
Ω(1) => no se cumple el primer if(), por tanto no itera en el while()
O(N) => N*L // Llamada a esURLDelimiter() en cada iteración

```
Void Tokenizador::MAIL(char*& p_izq, char*& p_der) const {
        if (p_izq != p_der){
                while (!parar && *p_der != ' ' && *p_der != '\0') {
                        if (*p_der != '@' || !EsDelimiter('@')) {
                                ++p_der;
                        } else { parar = true; }
                }
        }
        else if (p_izq == p_der && !EsDelimiter('@')) {
                while (*p_der != ' ' && *p_der != '\0') { ++p_der; }
        } else { ... }
}
```

Ω(1) => no se cumple las condiciones de los if, por tanto no llega a un bucle

O(N) => N+1itera el string en el primer while()

```
Void Tokenizador::Guion(char*& p_izq, char*& p_der) const {
        if (!EsDelimiter('-')) {GuionAux1(p_izq, p_der); }
        else GuionAux2(p_izq, p_der);
}
void Tokenizador::GuionAux1(char* &p_izq, char* &p_der) const {
        ...
        while (!EsDelimiter(*p_der) && *p_der != '\0') { ++p_der;}
}
void Tokenizador::GuionAux2(char* &p_izq, char* &p_der) const {
        if (EsDelimiter(*(p_der-1)) || EsDelimiter(*(p_der+1))) {
                ...
        } else {
                while (!parar && *p_der != ' ' && *p_der != '\0') {
                        if ((EsDelimiter(*(p_der-1)) ||EsDelimiter(*(p_der+1)))) {...}
                        else if (*p_der != '-' && EsDelimiter(*p_der)) { parar = true;}
                        else { ++p_der; }
                }
        }
}
```

N = caracteres que forman el subtring que está tratando

→ GuionAux1()
O(N) = N*L // En cada iteración del while() llama a EsDelimiter()

→ GuionAux2()
Ω(1) = 1 //no se cumple el if() por tanto no entra while()
O(N) = 1 + 1*(N+1+N) //Sumo N por la llamada al metodo EsDelimiter()

→ Guion()
Ω(1) = **N** + 1
O(N) = **N** + (1 + 1*(N+1+N))

```
void
Tokenizador::Generico(char*& p_der) const {
        ...
        while (!parar) {
                // IF: encuentra en caso especial THEN tokenizar especial
                // ELSE: seguir iterando
        }
}
```

N = length del string que se está tokenizando
X = complejidad del caso especial a ejecutar

Ω(N) => N+1 // Itera el string hasta encontrar un carácter que lo delimite
O(N) => N+1 + X //Dependerá del caso especial

```
void Tokenizador::Acronimo(char*& p_izq, char*& p_der) const {
        if (p_izq != p_der) {
                if (EsDelimiter('.')) AcronimoAux1(p_izq, p_der);
                else AcronimoAux2(p_izq, p_der);
        }
}
Void Tokenizador::AcronimoAux1(char* &p_izq, char*& p_der) const {
        if (EsDelimiter(*(p_der+1)) || EsDelimiter(*(p_der-1))) {
                ...
        } else {
                while (!parar && *p_der != '\0') {
                        parar  = ((*p_der == '.' && EsAcronimoDel(p_der+1))) || *p_der == ' ';
                        ++p_der;
                }
        }
}
void Tokenizador::AcronimoAux2(char* &p_izq, char*& p_der) const {
        if (EsDelimiter(*(p_der-1)) || EsDelimiter(*(p_der+1))) {
                ...
        } else {
                while (!parar && *p_der != '\0') {
                        parar  = ((*p_der == '.' && EsDelimiter(*(p_der+1)))) || *p_der == ' ';
                        ++p_der;
                }
        }
}
```

N = caracteres que forman el subtring que está tratando

→ AcronimoAux1() y AcronimoAux2()
Ω(1) = 1 //no se cumple el if() por tanto no entra while()
O(N) = (N+1)*L //N caracteres a recorrer * L al llamar EsAcronimoDel()

→ Acronimo()
Ω(1) = 1
O(N) = (N+1)*L + 1

```
void Tokenizador::Decimal(char*& p_izq, char*& p_der) const {
        if  (p_izq == p_der) { DecimalAux2(p_izq, p_der); }
        else  { DecimalAux1(p_izq, p_der); }
}
void Tokenizador::DecimalAux1(char*& p_izq, char*& p_der) const {
        while(*p_der != '\0' && *p_der != ' ' && !parar) {
                if (c == '.' || c == ',') {
                        if (*(p_der+1) == ' ') {...}
                        else if (!isdigit(*(p_der+1))){ Acronimo(p_izq, p_der); }
                        else { parar = false; ++p_der; }
                } else if (*p_der == 'E' && EsDelimiter(',')) {...}
                else if (EsDelimiter(*p_der)){...}
                else { parar = false; ++p_der; }
        }
}
```

N = caracteres que forman el subtring que está tratando

→DecimalAux1() y DecimalAux2()
Ω(1) = N
O(N) = N*N
→ Decimal()
O(N) = N*N

1.B Otros métodos usados en los casos especiales

```
Bool Tokenizador::EsDelimiter(const char& p_d) const{
        if (_delimitersLength > 0) {
                for (0...N-1) {
                        ...
                }
        }
}
```

N = _delimiters.length()
Ω(1) => No se cumple el if() o al entrar al bucle lo encuentra en la primera posición
O(N) => 1 + 1*(N+1)

```
const char* tr ="aaaaaaNNeeeeiiiiNñoooooNNuuuuNNNaaaaaaNNeeeeiiiiNñoooooNNuuuuNNN";
void Tokenizador::EliminarMinusAcentos (std::string& s) const {
        unsigned char a;
        char* c = &s.at(0);
        for (0...N-1) {...}
}
```

N = s.length()
f(EliminarMinusAcentos) = 1 + 1*(N+1)

```
std::string Tokenizador::ObtenerString(const char* p_i, const char* p_f) const {
        std::string s_out;
        if (p_i != p_f) { s_out.assign(p_i, (p_f – p_i)); }
        else { s_out.assign(p_i, 1); }
        return s_out;
}
```

N = (p_f – p_i) La resta de las posiciones de memoria final - inicial
f(string& assign const string& str) = N // Complejidad dada por la documentación
f(ObtenerString) = N + 1 // +1 por el if() y el retún

2. Cálculo de la complejidad espacial al tokenizar los ficheros

```
Bool Tokenizador::TokenizarListaFicheros(const std::string& p_in)  const {
   … // Por cada ruta de fichero
   while ((read = getline(&line, &len, fp)) != -1) {
     Tokenizar(line, line+".tk");
   }
    ...
}
```

Lruta = line.length() B // Tras leer una linea de la lista de ficheros
Lout = Lruta + ".tk"B
Lin = Lruta B
Memoria = (Lruta + Lout + Lin) = Lfichero * 3

```
Bool Tokenizador::Tokenizar (std::string& p_fin, std::string& p_fout) const {
   char* f = (char *) mmap (0, size, PROT_READ, MAP_PRIVATE, fd, 0);
   std::ofstream out(p_fout);
   char p = '\n';
   for (int i = 0; i < size; ++i) {
         char* pendline = f+i;
         while (*pendline != '\n') {
               ++pendline;
         }
         std::string Stokenizar = ObtenerString(f+i, pendline+1);
         TokenizarEspecial( Stokenizar, tokens );
         for (std::string s : tokens) {
               out.write(buffer, s.size());
         }
         i += pendline - (f+i);
   }
         ...
}
```

Lfichero = KB del fichero (se guarda en ram)
Ltokens = token[i].length de cada token
Memoria = LficheroKB + LtokensB