Estructures de la Informació. Control parcial

Dept. de Llenguatges i Sistemes Informàtics
E.P.S.E.V.G., 5 de novembre de 2013, 8:30-10:30

IMPORTANT: Resol els problemes en fulls separats.

1. (2 punts. 20 minuts) **Eficiència algorísmica**

   Donat aquest algorisme:

```
bool secret (char v[], int m, char c) {
  // pre: 0 < m <= mida(v)
  if (proc1(v, m-1, c)) return true;
  else return proc2(v, 0, m-1, c);
}

bool proc1 (char v[], int n, char c) {
  // pre: 0 < n < mida(v)
  int i = 1;
  bool b = v[0] == c;
  while ((i <= n) and not b) {
    if (v[i] == c) b = true;
    else i *= 2;
  }
  return b;
}

bool proc2 (char v[], int i, int j, char c) {
  // pre: 0 <= i <= j and j < mida(v)
  if (i==j) return false;
  else (i+1==j) return v[i]==c or v[j]==c;
  else return v[i] == c or v[j] == c or proc2(v, i+1, j-1, c);
}
```

   a) Calcula de manera **raonada** el cost de la funció `secret` respecte la mida *m* del vector. Si és necessari, pots utilitzar els teoremes adjunts.
   b) Hi ha alguna manera més eficient de fer el mateix tenint en compte que el vector estigués ordenat? Depenent de la resposta, descriu la solució o raona per què no.

Decreixement aritmètic

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n-c)+g(n) & si\ n_0\leqslant n \end{cases}$$

on n_0 és una constant, $c\geqslant 1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb *k* constant. Llavors,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<1 \\ \Theta\left(n^{k+1}\right) & si\ a=1 \\ \Theta\left(a^{n/c}\right) & si\ a>1 \end{cases}$$

Decreixement geomètric

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n/b)+g(n) & si\ n_0\leqslant n \end{cases}$$

on n_0 és una constant, $b>1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb *k* constant. Llavors,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<b^k \\ \Theta\left(n^k\log n\right) & si\ a=b^k \\ \Theta\left(n^{\log_b a}\right) & si\ a>b^k \end{cases}$$

2. (5 punts. 60 minuts) **Estructures Lineals**

Donada la següent classe llista:

```
class llista {
   ...
   void int2list(int n) throw(error);
   ...
 private:
   // Llista simplement enllaçada, circular i sense element fantasma.
   struct node {
     int info;
     node *seg;
   };
   node* _head;
   int _size;
};
```

Implementa en C++ el mètode `int2list` de l'anterior classe llista que donat un nombre enter positiu *n* crea una llista amb tants nodes com dígits té *n,* on a cada node de la llista es guarda de manera consecutiva un dígit de *n*.

Per exemple, si féssim la crida `L.int2list(17310)` crearia la llista:

```
L={1, 7, 3, 1, 0}
```

Cal tenir present que l'atribut `_head` sempre apunta a l'últim element de la llista. En aquest exemple apuntaria al node amb l'element 0.

NOTA: Cal que implementis els mètodes addicionals que utilitzis explícitament.

3. (3 punts. 40 minuts) **Arbres**

Donada la representació de la classe arbre general:

```
private:
  struct node {
    T info;
    node *primf;
    node *seggerm;
  };
  node *_arrel;
```

a) Implementa en C++ un mètode `print` de la classe arbre general que mostra la informació de l'arbre per la sortida estàndard. Aquesta informació s'ha de mostrar de manera que:
   - cada node de l'arbre s'imprimeix en una línia,
   - els nodes fills d'un node donat sempre es mostren a continuació d'aquest node,
   - cada node té una identació proporcional a la fondària del node.

b) Quin és el cost de la teva implementació? Es pot fer una implementació més eficient? Raona la teva resposta.

NOTA: Cal que implementis els mètodes addicionals que utilitzis explícitament.

Solució problema 1

a)

La funció `secret` té una crida a `proc1()` i una altra crida a `proc2()`. Per tant cal calcular el cost d'aquestes dues funcions.

Càlcul del cost temporal de la funció `proc1()`:

Aquesta funció està composada de diverses operacions de cost constant i un bucle.
El cost temporal del bucle a diferència del que podria semblar no és *n*, ja que el nombre d'iteracions del bucle no són *n*. La variable *i* s'incrementa en cada iteració multiplicant per dos. Per tant, aquest bucle tindrà *log n* iteracions i cost $\Theta(\log n)$.
El cost temporal de `proc1()` és el màxim de tots els costos i per tant és $\Theta(\log n)$.

Càlcul del cost temporal de la funció `proc2()`:

Sigui T(n) el cost d'aquesta funció. Si n>=2, es fan operacions de cost constant i una crida recursiva que rep *n-2* d'elements. Per tant, es pot utilitzar el teorema mestre de decreixement aritmètic:

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<2 \\ a\cdot T(n-c)+\Theta(n^k) & si\ 2\leqslant n \end{cases}$$

Tenint en compte que *a*=1 (només hi ha una crida), *c*=2 (decrementem en dos unitats les dades en cada crida), *k*=0 (la resta d'operacions són de cost constant), l'equació de decreixement és la següent:

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<2 \\ T(n-2)+\Theta(1) & si\ 2\leqslant n \end{cases}$$

Donat que a=1 el cost temporal de `proc2()` és

$$T(n)=\Theta(n^{k+1})=\Theta(n^{0+1})=\Theta(n)$$

Càlcul del cost temporal de la funció `secret()`:

El cost temporal d'aquesta funció és el màxim del cost de les anteriors funcions. Per tant el cost final és $\Theta(n)$.

b)

L'operació secret el que fa és indicar si un element es troba en la taula. Per tant, si el vector estigués ordenat es podria aplicar un algorisme més eficient com ara l'algorisme de cerca dicotòmica que té cost $\Theta(\log n)$ on *n* és el nombre d'elements del vector.

Aquest algorisme comença comparant l'element a buscar amb un element qualsevol de la taula (normalment l'element central) que anomenarem element triat, i es poden donar tres casos:

- ja hem trobat l'element, per tant s'acaba la cerca

- el valor de l'element triat és major que el de l'element cercat, per tant es repeteix el procediment en la part de la taula que va des de l'inici fins l'element triat.
- en cas contrari es pren la taula que va des de l'element triat fins el final.

D'aquesta forma obtenim cada cop intervals més petits fins a arribar a un interval indivisible. Si l'element cercat no es troba dins d'aquest últim interval llavors es pot dir que l'element no es troba a l'array.

Solució problema 2

```
void llista::int2list(int n) throw(error) {
  // Pre: n>=0
  node *final;
  if (n == 0) {
    final = new node;
    final->info = 0;
    final->seg = final;
    _size = 1;
  }
  else {
    node *nou = NULL;
    int i = 0;
    while (n != 0) {
      try {
        if (i == 0) {
          final = new node;
          final->info = n%10;
          final->seg = NULL;
          nou = final;
        }
        else {
          node *p = new node;
          p->info = n%10;
          p->seg = nou;
          nou = p;
        }
      }
      catch (error) {
        esborra_nodes(nou);
        throw;
      }
      n = n/10;
      ++i;
    }
    final->seg = nou;
    _size = i;
  }
  esborra_nodes_circ(_head);
  _head = final;
}

// Mètode privat de classe. Caldria declarar-lo com a static dins la part
// privada de l'especificació de la classe (.hpp).
void llista::esborra_nodes(node *n) throw() {
  while (n != NULL) {
    node *p = n->seg;
    delete n;
    n = p;
  }
}

// Mètode privat de classe.
void llista::esborra_nodes_circ(node *n) throw() {
  while (n != NULL and n->seg != n) {
    node *p = n->seg->seg;
    delete n->seg;
    n->seg = p;
  }
  if (n != NULL) delete n;
}
```

Solució problema 3

a)

Una possible solució podria ser la següent:

```
template <typename T>
void Arbre<T>::print () const throw() {
  cout << _arrel->info << endl;
  return print(_arrel->primf, 1);
}

template <typename T>
void Arbre<T>::print (node *n, int ident) const throw() {
  if (n != NULL) {
    for (int i=0; i < ident; ++i) {
      cout << "\t";
    }
    cout << n->info << endl;
    print(n->primf, ident+1);
    print(n->seggerm, ident);
  }
}
```

Encara que hi hauria una altra forma que seria usant la classe string per així poder escriure directament les identacions:

```
template <typename T>
void Arbre<T>::print () const throw() {
  return print(_arrel, "");
}

template <typename T>
void Arbre<T>::print (node *n, const string &ident) const throw() {
  if (_arrel != NULL) {
    cout << ident;
    cout << n->info << endl;
    print(n->primf, ident+"\t");
    print(n->seg_germa, ident);
  }
}
```

b)

El cost temporal mig de la solució plantejada és **linial** ja que visita tots els nodes de l'arbre un sol cop. Però, cal tenir present que cost temporal del cas pitjor de la primera solució plantejada (l'arbre té forma de llista encadenada) té cost quadràtic.

El cost temporal mig com a mínim per qualsevol implementació d'aquest mètode és $\Theta(n)$ ja que aquest mètode ha de visitar tots els elements de l'arbre per poder-los mostrar.

Algorismes extres per algun altre exercici d'eficiència

```
bool proc1 (int v[], int i, int j, int c) {
  // pre: n > 0 and v ordenat creixentment
  int k = (j-i)/2;
  if (v[k] == c) return true;
  else
    if (v[k] < c) return proc1(v, i, k, c);
    else return proc1(v, k, j, c);
}

bool proc2 (int v[], int i, int j, int c) {
  // pre: i < j
  while ((i != j) and (i+1 != j) and (v[i] != c) and (v[j] != c)) {
    ++i;
    --j;
  }
  return i!=j and (v[i] == c or v[j] == c);
}
```