Corectitudinea algoritmilor

21 noiembrie 2016

1 Introducere

Un algoritm trebuie analizat din cel puțin două puncte de vedere:

- Eficiență (complexitate): cantitatea de resurse (timp și memorie) solicitate.
- Corectitudine: pe orice date de intrare corecte (care respectă un set de precondiții) algoritmul întoarce rezultate corecte (care respectă un set de postcondiții).

Pentru a demonstra acest "orice" este nevoie de o analiză formală, bazată pe următorii pași:

- Idenficarea precondițiilor algoritmului. Totalitatea acestora constituie **aserțiunea de intrare** $P_I : P_I : I- > \{0, 1\}$ (I = domeniul datelor de intrare)
- Identificarea postcondițiilor algoritmului. Totalitatea acestora constituie **aserțiunea de ieșire** $P_O : P_O : I \times O- > \{0, 1\}$ (O = domeniul rezultatelor)
- Demonstrația că, plecând din starea inițială a algoritmului, în care se respectă aserțiunea de intrare, pașii descriși de algoritm ne duc într-o stare finală în care se respectă aserțiunea de ieșire.

Când demonstrăm și că algoritmul se termină într-un număr finit de pași, demonstrăm că algoritmul este **total corect**: $\forall d \in I \bullet P_I(d) \Rightarrow (termin(Alg(d)) \wedge P_O(d, Alg(d)))$. Alternativa este să demonstrăm că algoritmul este **parțial corect**: $\forall d \in I \bullet (P_I(d) \wedge termin(Alg(d))) \Rightarrow P_O(d, Alg(d))$.

Ideea demonstrației este să identificăm stările intermediare prin care ar trebui să treacă algoritmul și să ne asigurăm că fiecare acțiune efectuează trecerea dintr-o asemenea stare în starea următoare. Demonstrația este în general simplă pe algoritmi secvențiali, și ceva mai dificilă în cazul algoritmilor recursivi sau care conțin bucle. Construirea unei asemenea demonstrații poate ajuta la identificarea și corectarea erorilor.

2 Invarianți la ciclare

2.1 Definiție

Cea mai frecventă tehnică de demonstrare a corectitudinii algoritmilor recursivi, respectiv iterativi, este inducția. În cazul algoritmilor iterativi, metoda constă în identificarea (și demonstrarea) de **invarianți la ciclare**: proprietăți valabile la intrarea într-o buclă și conservate pe durata și la ieșirea din buclă.

Invarianții la ciclare se demonstrează în 3 pași (corespunzând, practic, unei inducții matematice după variabila de ciclare):

- Inițializare: invariantul se respectă înainte de prima intrare în buclă (iterația 0).
- Menținere: dacă invariantul se respectă înainte de iterația k, atunci el se va respecta și după iterația k (înainte de iterația k+1).
- Terminare: invariantul se respectă la ieșirea definitivă din buclă, ceea ce reprezintă o proprietate utilă în demonstrarea corectitudinii algoritmului.

2.2 Exemple

2.2.1 Suma elementelor dintr-un vector

```
suma(v, n) {
  s = 0;
  i = 0;
  while (i < n) {
    s = s + v[i];
    i++;
  }
}
```

Invariant: $s_i = \sum_{j=0}^{i-1} v[j]$.

Inițializare: Înainte de prima iterație, $s_0 = 0 = \sum_{j=0}^{-1} v[j]$.
Menținere: Presupunem că înainte de iterația i $s_i = \sum_{j=0}^{i-1} v[j]$.
După iterația i (înainte de iterația i+1) vom avea $s_{i+1} = \sum_{j=0}^{i-1} v[j] + v[i] = \sum_{j=0}^{i} v[j]$.
Terminare: La ieșirea definitivă din buclă (înainte de iterația n): $s_n = \sum_{j=0}^{n-1} v[j]$, adică suma tuturor elementelor din vector.

2.2.2 Sortarea prin selecție

```
sel-sort(v, n) {
  for (i=0; i<n-1; i++) {
    min = i;
    for (j=i+1; j<n; j++)
      if v[j] < v[min]
        min = j
    swap(v[i], v[min])
  }
}
```

Invariant pentru bucla interioară: Înainte de iterația j, min_j = indexul elementului minim din $v[i..j-1]$.

Inițializare: Înainte de prima iterație, $min_{i+1} = i$ = indexul elementului minim din $v[i..i]$.
Menținere: Presupunem că înainte de iterația j min_j = indexul elementului minim din $v[i..j-1]$.
După iterația j (înainte de iterația j+1), dacă v[j] < min_j, atunci $min_{j+1} = v[j]$, altfel $min_{j+1} = min_j$.
Așadar, min_{j+1} = indexul elementului minim din $v[i..j]$.
Terminare: La ieșirea definitivă din buclă (înainte de iterația n): min_n = indexul elementului minim din v[i..n-1].

Invariant pentru bucla exterioară: Înainte de iterația i, $v[0..i-1]$ sortat și conține cele mai mici i elemente din v.

Inițializare: Înainte de prima iterație, v[0..-1] sortat și conține cele mai mici 0 elemente din v.
Menținere: Presupunem că înainte de iterația i $v[0..i-1]$ sortat și conține cele mai mici i elemente din v.
După iterația i (înainte de iterația i+1), v[0..i-1] rămâne nemodificat, iar v[i] primește valoarea v[min], unde min = indexul elementului minim din v[i..n-1] (conform invariantului anterior).
Așadar, înainte de iterația i+1, v[0..i] sortat și conține cele mai mici i+1 elemente din v.
Terminare: La ieșirea definitivă din buclă (înainte de iterația n-1): v[0..n-2] sortat și conține cele mai mici n-1 elemente din v. Rezultă că v[n-1] conține cel mai mare element din v, deci v[0..n-1] sortat.

Exerciții

1. Căutarea binară:

```
bin-search(v, x, start, stop) {
  while (start <= stop) {
    mid = (start+stop)/2;
    if (x == v[mid])
      return mid;
    if (x > v[mid])
      start = mid+1;
    else
      stop = mid-1;
  }
  return null;
}
```

2. Sortarea prin inserție:

```
ins-sort(v, n) {
  for (j=1; j<n; j++) {
    x = v[j];
    i = j-1;
    while (i>=0 && v[i]>x) {
      v[i+1] = v[i];
      i--;
    }
    v[i+1] = x;
  }
}
```

3. Sortarea prin metoda bulelor:

```
bubble-sort(v, n) {
  for (i=0; i<n; i++)
    for (j=n-1; j>i; j--)
      if v[j] < v[j-1]
        swap(v[j], v[j-1])
}
```

4. Funcția de partiționare din cadrul algoritmului quicksort:

```
partition(v, start, stop) {
  x = v[stop];
  i = start-1;
  for (j=start; j<stop; j++)
    if v[j] < x {
      i++;
      swap(v[i], v[j]);
    }
  swap(v[i+1], v[stop])
  return i+1;
}
```

5. Determinarea distanței minime prin algoritmul lui Dijkstra:

```
dijkstra(g, c, s, adj) {
  foreach v in vertices(g)
    d[i] = infinity;
  d[s] = 0;
  q = vertices(g);
  while q {
    u = extract_min(q);
    q.remove(u);
    foreach v in adj[u]
      if d[u] + c[u,v] < d[v]
        d[v] = d[u] + c[u,v]
  }
}
```