Reasoning About Algorithms

Hendrik Speleers

Reasoning About Algorithms

- Overview
  - Reasoning about iteration
  - Reasoning about algorithm design
  - Reasoning about complexity
  - Examples
    - Summation, integer division, power
    - Searching in arrays: linear search, binary search
    - Sorting: selection sort, insertion sort
    - Maximum subarray sum problem

Reasoning About Algorithms

- Reasoning about iteration
  - What we have seen before: invariant to prove correctness of loop

```
<initialization>
// invariant
while (condition) {
    // invariant && condition
    <statements>
    // invariant
}
// invariant && !condition
// => result
```

invariant: a property that is true before and after each iteration

result: a consequence of invariant and negation of condition

weak condition gives robust result

Reasoning About Algorithms

- Reasoning about iteration
  - What we have seen before: summation

```
int s = 0, n = 0;

// inv: ??
while (n < 20) {

    n += 1;

    s += n;

}

// res: s == sum(1..20)
```

condition

invariant && !condition

Reasoning About Algorithms

Università di Roma Tor Vergata

- Reasoning about iteration
  - What we have seen before: summation

```
int s = 0, n = 0;

// inv: s == sum(1..n) && n <= 20
while (n < 20) {
    // s == sum(1..n)   && n < 20
    n += 1;
    // s == sum(1..n-1) && n <= 20
    s += n;
    // s == sum(1..n)   && n <= 20
}
// s == sum(1..n) && n <= 20 && n >= 20
// res: s == sum(1..20)
```

Reasoning About Algorithms

Università di Roma Tor Vergata

- Reasoning about algorithm design
  - Stepwise design of iterative algorithm containing a loop
    - Specification of precondition and postcondition (result)
    - Solution strategy for algorithm
    - Specification of invariant
    - Proof of correctness by invariant
      - Initialization of invariant
      - Maintenance of invariant
      - Termination of algorithm: correct result in finite number of iterations
  - Example: computation of integer division
    - Input: dividend (x) and divisor (y), positive integers
    - Output: quotient (q) and remainder (r)

Reasoning About Algorithms

- Reasoning about algorithm design
  - Computation of integer division

```
// pre: (x >= 0) && (y > 0)

int q, r;

while (   ) {

    r =

    q =

}
// res: (q * y + r == x) && (r >= 0 && r < y)
```

precondition

result

Reasoning About Algorithms

- Reasoning about algorithm design
  - Computation of integer division

```
// pre: (x >= 0) && (y > 0)

int q, r;
// inv: (q * y + r == x)   && (r >= 0)
while (   ) {

    r -= y;

    q += 1;

}
// res: (q * y + r == x)   && (r >= 0 && r < y)
```

invariant

strategy

Reasoning About Algorithms

- Reasoning about algorithm design
  - Computation of integer division

```
// pre: (x >= 0) && (y > 0)

int q = 0, r = x;
// inv: (q * y + r == x)   && (r >= 0)
while (r >= y) {

   r -= y;

   q += 1;

}
// res: (q * y + r == x)   && (r >= 0 && r < y)
```

initialization

termination

Reasoning About Algorithms

- Reasoning about algorithm design
  - Computation of integer division

```
// pre: (x >= 0) && (y > 0)

int q = 0, r = x;
// inv: (q * y + r == x)    && (r >= 0)
while (r >= y) {
   // (q * y + r == x)      && (r >= y > 0)
   r -= y;
   // (q * y + y + r == x)  && (r >= 0)
   q += 1;
   // (q * y + r == x)      && (r >= 0)
}
// res: (q * y + r == x)    && (r >= 0 && r < y)
```

complete proof

Reasoning About Algorithms

- Reasoning about complexity
  - Example: computation of powers
    - Powers by repeated multiplication

$$x^n = \overbrace{\underbrace{x * x * \ldots * x}_{k \text{ times}} * x * x * \ldots * x}^{n \text{ times}}$$

- Input: base (x, int), exponent (n, int); Output: power (p)
- Version 1: $p = x^k$ $k = 0 \gg n$
- Version 2: $x^n = x^k * p$ $k = 0 \ll n$

Reasoning About Algorithms

Università di Roma
Tor Vergata

- Reasoning about complexity
  - Example: computation of powers
    - Powers by repeated multiplication

$$x\hat{}n \;=\; \overbrace{\underbrace{\boxed{x * x * \ldots * x}}_{k \text{ times}} * \boxed{x * x * \ldots * x}}^{n \text{ times}}$$

- Input: base (x, int), exponent (n, int); Output: power (p)
- Version 1: $(x*p) = x\hat{}(k+1)$ $\qquad k = 0$ » n
- Version 2: $x\hat{}n = x\hat{}(k-1) * (x*p)$ $\qquad k = 0$ « n

Reasoning About Algorithms

Università di Roma Tor Vergata

- Reasoning about complexity
  - Computation of powers: version 1

```
// pre: n >= 0
int k = 0, p = 1;
// inv: p == power(x,k) && k <= n
while (k < n) {

   k += 1;

   p *= x;

}

// res: p == power(x,n)
```

Reasoning About Algorithms

Università di Roma Tor Vergata

- Reasoning about complexity
  - Computation of powers: version 1

```
// pre: n >= 0
int k = 0, p = 1;
// inv: p == power(x,k) && k <= n
while (k < n) {
    // p == power(x,k)   && k < n
    k += 1;
    // p == power(x,k-1) && k <= n
    p *= x;
    // p == power(x,k)   && k <= n
}
// p == power(x,k) && k <= n && k >= n
// res: p == power(x,n)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Computation of powers: version 2

```
// pre: n >= 0
int k = n, p = 1;
// inv: p * power(x,k) == power(x,n) && k >= 0
while (k > 0) {

   k -= 1;

   p *= x;

}

// res: p == power(x,n)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Computation of powers: version 2

```
// pre: n >= 0
int k = n, p = 1;
// inv: p * power(x,k) == power(x,n) && k >= 0
while (k > 0) {
   // p * power(x,k) == power(x,n)   && k > 0
   k -= 1;
   // p * power(x,k+1) == power(x,n) && k >= 0
   p *= x;
   // p * power(x,k) == power(x,n)   && k >= 0
}
// p * power(x,k) == power(x,n) && k == 0
// res: p == power(x,n)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Computation of powers: more efficient ?
    - Version 2

      x^n = x^($k-1$) * (x*p)

      in total: n multiplications

    - Version 3

      k even : x^n = (x*x)^($k/2$) * p

      k odd : x^n = x^($k-1$) * (x*p)

      reduction in computational cost

Reasoning About Algorithms

- Reasoning about complexity
  - Computation of powers: version 3

```
// pre: n >= 0
int k = n, y = x, p = 1;
while (k > 0) {
    if (k % 2 == 0) { // k even
        k /= 2;
        y *= y;
    } else {          // k odd
        k -= 1;
        p *= y;
    }
}
// res: p == power(x,n)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Computation of powers: version 3

```
// pre: n >= 0
int k = n, y = x, p = 1;
// inv: p * power(y,k) == power(x,n) && k >= 0
while (k > 0) {
   // p * power(y,k) == power(x,n)   && k > 0
   if (k % 2 == 0) { // k even
      ... // p * power(y,k) == power(x,n) && k >= 0
   } else {           // k odd
      ... // p * power(y,k) == power(x,n) && k >= 0
   }
} // p * power(y,k) == power(x,n) && k == 0
// res: p == power(x,n)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Computation of powers: version 3

```
...
// p * power(y,k) == power(x,n)        && k > 0
if (k % 2 == 0) { // k even
   // p * power(y,k) == power(x,n)     && k > 0
   k /= 2;
   // p * power(y,2*k) == power(x,n)   && k > 0
   y *= y;
   // p * power(y,k) == power(x,n)     && k > 0
} else {             // k odd
   ...
}
...
```

Reasoning About Algorithms

Università di Roma Tor Vergata

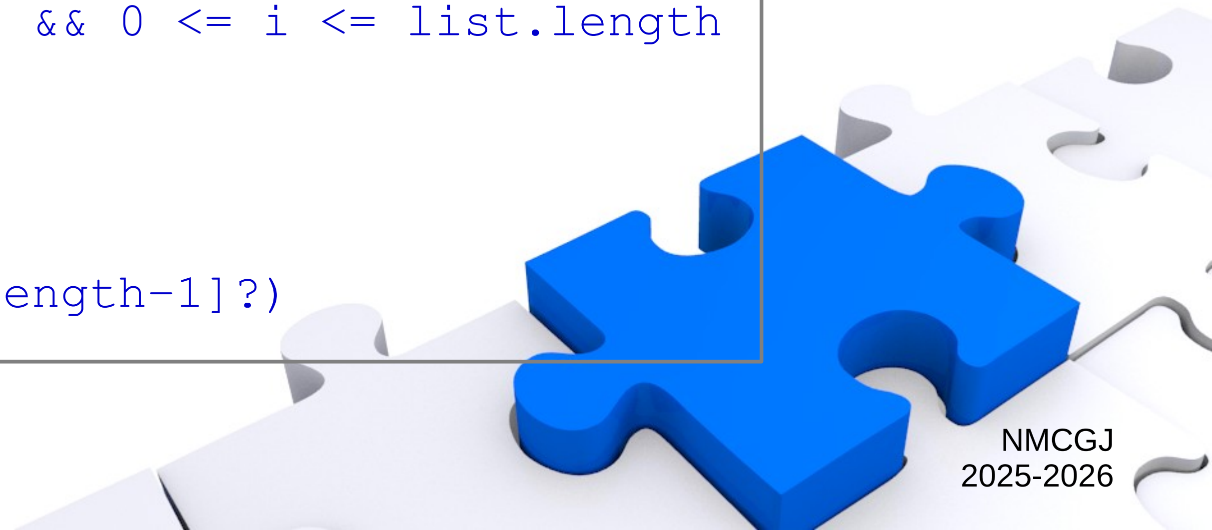
- Reasoning about complexity
  - Computation of powers: version 3

```
...
// p * power(y,k) == power(x,n)       && k > 0
if (k % 2 == 0) { // k even
    ...
} else {          // k odd
    // p * power(y,k) == power(x,n)   && k > 0
    k -= 1;
    // p * power(y,k+1) == power(x,n) && k >= 0
    p *= y;
    // p * power(y,k) == power(x,n)   && k >= 0
}
...
```

Reasoning About Algorithms

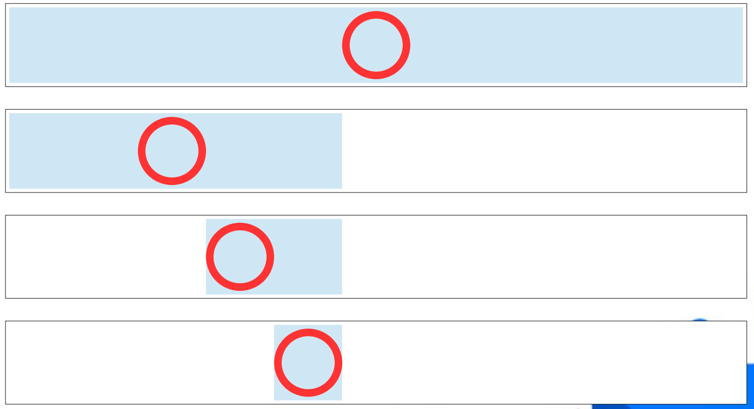
- Reasoning about complexity
  - Example: searching in arrays
    - Version 1: linear search
      - Running over elements in array (*list*)
      - Comparing current element with given value (*v*)

```
int i = 0; boolean found = false;
// inv: found == (v in list[0..i-1]?) && 0 <= i <= list.length
while (i < list.length && !found) {
    found = (list[i] == v);
    i++;
}
// res: found == (v in list[0..list.length-1]?)
```



Reasoning About Algorithms

- Reasoning about complexity
  - Example: searching in arrays
    - Version 2: binary search
      - Assuming a sorted array (*list*), from small to large values
      - Comparing any element with given value (*v*)
        - If larger, then not after
        - If smaller, then not before



      - Invariant
        - nonempty subarray that has to be looked in

Reasoning About Algorithms

Università di Roma Tor Vergata

- Reasoning about complexity
  - Searching in arrays: binary search

```
// pre: sorted list
int a = 0, b = list.length - 1;
while (a < b) {
    int m = (a + b) / 2;
    if (v <= list[m]) { // smaller or equal
        b = m;
    } else {            // larger
        a = m + 1;
    }
}
boolean found = (list[a] == v);
// res: found == (v in list[0..list.length-1]?)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Searching in arrays: binary search

```
// pre: sorted list
int a = 0, b = list.length - 1;
// inv: a <= b   && (v in list[a..b]? || v not in list?)
while (a < b) {
   // a < b       && (v in list[a..b]? || v not in list?)
   int m = (a + b) / 2;
   // a <= m < b && (v in list[a..b]? || v not in list?)
   ...

} // a == b && (v in list[a..b]? || v not in list?)
boolean found = (list[a] == v);
// res: found == (v in list[0..list.length-1]?)
```

Reasoning About Algorithms

- Reasoning about complexity
  - Searching in arrays: binary search

```
...
int m = (a + b) / 2;
// a <= m < b && (v in list[a..b]? || v not in list?)
if (v <= list[m]) { // smaller or equal
    // a < b  && (v in list[a..m]? || v not in list?)
    b = m;
    // a <= b && (v in list[a..b]? || v not in list?)
} else {                // larger
    // a < b  && (v in list[m+1..b]? || v not in list?)
    a = m + 1;
    // a <= b && (v in list[a..b]? || v not in list?)
} ...
```

Reasoning About Algorithms

- Reasoning about complexity
  - Analysis of complexity
    - An indication of efficiency of an algorithm (~ computer effort)
    - Time complexity (speed) and space complexity (memory)
    - Big-O notation
      - *O(1)*, *O(n^k)*, *O(log(n))* for problem-size *n*
  - Time complexity in our examples:
    - Power: version 1/2 is *O(n)*
      version 3 is *O(log(n))*
    - Searching in arrays: linear search is *O(n)* on average
      binary search is *O(log(n))* on average

Reasoning About Algorithms

- Sorting algorithms
  - Sorting an array of comparable objects (size *n*)
    - **Selection sort**
    - **Insertion sort**

    - Bubble sort
    - Quick sort
    - Merge sort
    - Heap sort
    - Shell sort
    - Radix sort
    - ...

Reasoning About Algorithms

Università di Roma Tor Vergata

- Sorting algorithms
  - Selection sort
    - Input: array (*list*) of size *n* = *list.length*
    - Algorithm in pseudo-code

```
for i from n−1 to 1 do
    find position of maximum in list[0..i]
    swap maximum and element list[i]
end for
```

    - DEMO

Reasoning About Algorithms

- Sorting algorithms
  - Selection sort

```
for (int i = list.length-1; i >= 1; i--) {
   // sorted(list[i+1..n-1])?
   int pos = i, max = list[i];
   for (int j = i-1; j >= 0; j--)
      if (list[j] > max) {
         pos = j; max = list[j];
      }
   list[pos] = list[i];
   list[i] = max;
   // sorted(list[i..n-1])? && list[i] >= list[0..i-1]
}
// res: sorted(list[0..n-1])
```

Reasoning About Algorithms

- Sorting algorithms
  - Selection sort
    - Complexity: counting most time-consuming operations
      - Basic mathematical operations (*MO*): +, −, / , *
      - Array element operations (*AO*) : assignments + comparisons involving arrays
    - Array element operations
      - $AO_{min} = 3\ (n-1)\ + ((n-1) + (n-2) + \ldots + 1)$ « array already sorted
        $= 3\ (n-1) + n\ (n-1) / 2$
      - $AO_{max} = 3\ (n-1) + 2\ ((n-1) + (n-2) + \ldots + 1)$
        $= 3\ (n-1) + n\ (n-1)$
      - $AO_{avg} = (AA_{min} + AA_{max}) / 2 = O(n\hat{}2)$

complexity is *O(n^2)*

Reasoning About Algorithms

- Sorting algorithms
  - Insertion sort
    - Input: array (*list*) of size *n* = *list.length*
    - Algorithm in pseudo-code

```
for i from 1 to n−1 do
    insert list[i] sorted in list[0..i−1]
end for
```

- DEMO

Reasoning About Algorithms

- Sorting algorithms
  - Insertion sort

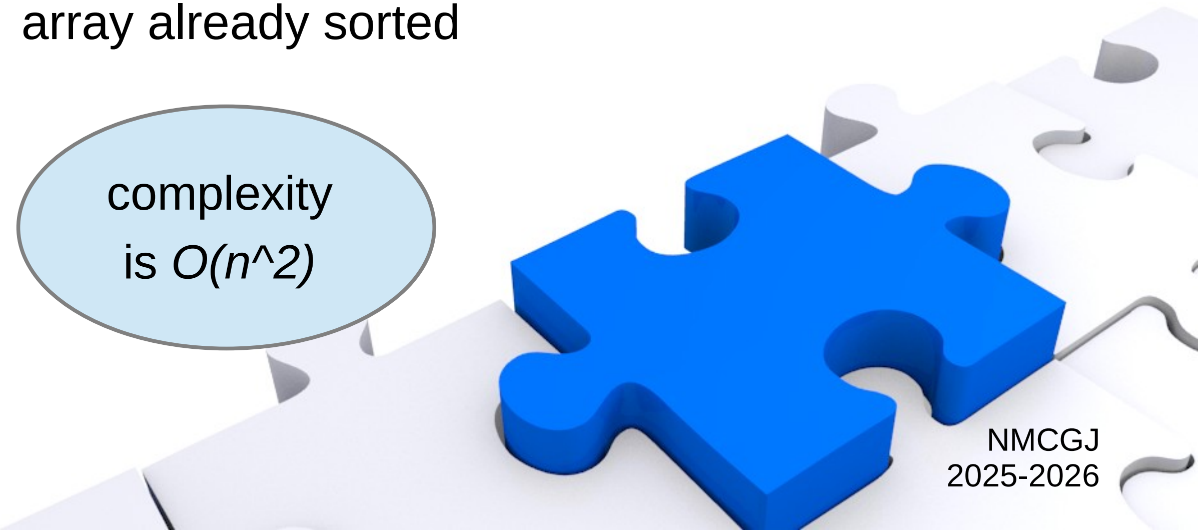
```
for (int i = 1; i < list.length; i++) {
   // sorted(list[0..i-1])?
   int j = i - 1, val = list[i];
   while (j >= 0 && list[j] > val) {
      list[j+1] = list[j];
      j--;
   }
   list[j+1] = val;
   // sorted(list[0..i])?
}
// res: sorted(list[0..n-1])
```

Reasoning About Algorithms

Università di Roma Tor Vergata

- Sorting algorithms
  - Insertion sort
    - Complexity: counting most time-consuming operations
      - Basic mathematical operations (*MO*): +, −, / , *
      - Array element operations (*AO*) : assignments + comparisons involving arrays
    - Array element operations
      - $AO_{min} = 3\ (n-1)$ « array already sorted
      - $AO_{max} = 2\ (n-1) + n\ (n-1)$
      - $AO_{avg} = (AA_{min} + AA_{max})\ /\ 2 = O(n\text{^}2)$

complexity is *O(n^2)*



Reasoning About Algorithms

Università di Roma Tor Vergata

- Maximum subarray sum problem
  - Finding the contiguous subarray within an array which has largest sum
  - Version 1
    - Input: array (*list*) of size *n* = *list.length*,   Output: max subarray sum (*max*)
    - Algorithm in pseudo-code

```
for each subarray
    compute its sum and check if it is the largest so far
end for
```

- DEMO

Reasoning About Algorithms

- Maximum subarray sum problem
  - Version 1

```
int max = list[0];          // max sum
for (int a = 0; a < list.length; a++) {
    int maxa = list[a];     // max sum starting from a
    for (int b = a; b < list.length; b++) {
        int sum = list[a]; // sum of list[a..b]
        for (int i = a+1; i <= b; i++)
            sum += list[i];
        maxa = Math.max(maxa, sum);
    }
    max = Math.max(max, maxa);
}
// res: max == max_sub_sum(list[0..n-1])
```

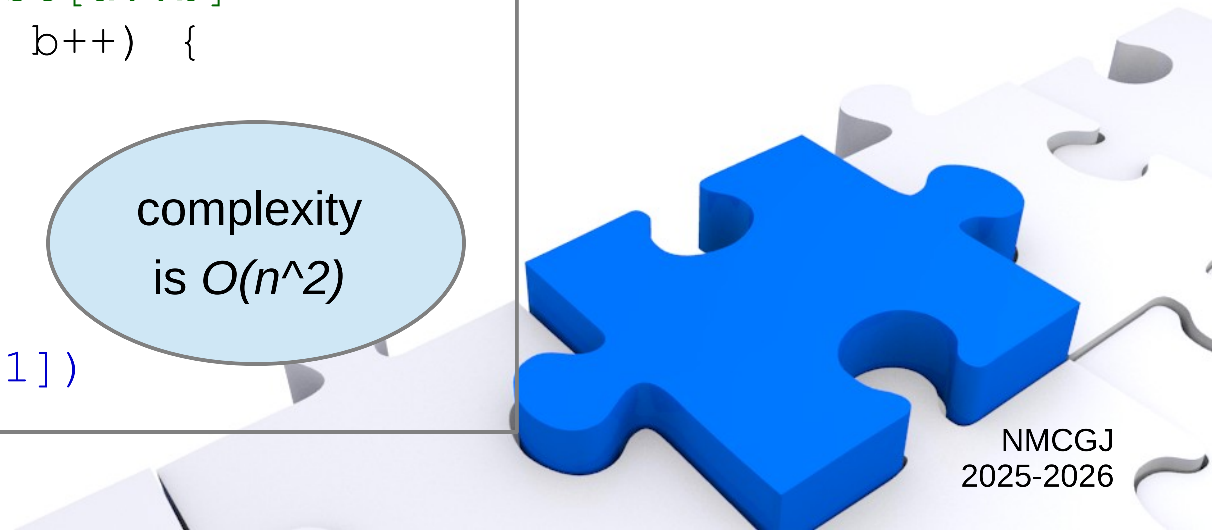
complexity is $O(n\text{\^{}}3)$

Reasoning About Algorithms

- Maximum subarray sum problem
  - Version 2
    - Removal of inner for loop

```
int max = list[0];          // max sum
for (int a = 0; a < list.length; a++) {
    int maxa = list[a];     // max sum starting from a
    int sum = list[a];      // sum of list[a..b]
    for (int b = a+1; b < list.length; b++) {
        sum += list[b];
        maxa = Math.max(maxa, sum);
    }
    max = Math.max(max, maxa);
}
// res: max == max_sub_sum(list[0..n-1])
```

complexity is *O(n^2)*



Reasoning About Algorithms

- Maximum subarray sum problem
  - Version 2 (bis)
    - Reordering of subarrays

```
int max = list[0];          // max sum
for (int a = 0; a < list.length; a++) {
    int maxa = list[a];     // max sum ending at a
    int sum = list[a];      // sum of list[b..a]
    for (int b = a-1; b >= 0; b--) {
        sum += list[b];
        maxa = Math.max(maxa, sum);
    }
    max = Math.max(max, maxa);
}
// res: max == max_sub_sum(list[0..n-1])
```

complexity is $O(n^2)$

Reasoning About Algorithms

Università di Roma Tor Vergata

- Maximum subarray sum problem
  - Version 3 (Kadane's algorithm)
    - Avoiding inner for loop by accumulating the sum *maxa*

```
int max = list[0];        // max sum
int maxa = list[0];       // max sum ending at a
for (int a = 1; a < list.length; a++) {
   maxa = Math.max(0, maxa) + list[a];
   max = Math.max(max, maxa);
}
// res: max == max_sub_sum(list[0..n-1])



// proof of correctness as an excercise
```

complexity is $O(n)$