Faculty Development Program

Design and Analysis of Algorithms

Introduction to Algorithms

R. S. Milton
Department of Computer Science and Engineering
SSN College of Engineering

11 December, 2014

Contents

1. Outcomes

At the end of the course, the student will be able to:

- design algorithms for various computing problems,
- analyze the time and space complexity of algorithms,
- critically analyze the different algorithm design techniques for a given problem.
- modify existing algorithms to improve efficiency.

Books

- Anany Levitin, "Introduction to the Design and Analysis of Algorithms", Third Edition, Pearson Education, 2012.
- S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, "Algorithms", McGraw-Hill Higher Education, 2006

2. Algorithmic Problem Solving

3. Important Problem Types

4. What is an Algorithm?

Algorithm

A well-defined computational procedure composed of of instructions that solves the problem in a finite number of steps.

- A computational problem is specified by its input, output, and the relation between the input and the output.
- Algorithms are procedural solution to problems.

Problem, Process, Algorithm

- A computational probelm is defined by the input given, the output required and the input–output relation (precondition and the postcondition).
- A computational process starts with the precondition, evolves, and terminates with the postcondition.
- A computational process is generated by an algorithm. When an algorithm is executed, a process evolves.

5. Design, correctness, efficiency

- Specify the problem precisely: its input, output, and input–output relation.
- Design an algorithm for solving a problem.
- Prove the correctness of the algorithm: when the algorithm is executed, the specified input–output relation is satisfied
- Analyze the efficiency of the algorithm: the resources (running time and memory space) required by the algorithm to finish execution.

6. Problem specification

> Speed takes you nowhere if you are heading in the wrong direction.
>
> American proverb

- Understand the computational process (or invent a process).
- Execute the process with examples.
- Specify the computational problem in terms of its input, output, and the relation between the input and the output.
- Is there an algorithm in the manual?
- If there area many, compare algorithms.

7. Computing constraints

- Computing model: von Neumann machine, RAM random access machine $\rightsquigarrow$ sequential algorithms
- Resources available vs huge volumes of data and time-criticality
- Exact vs approximate algorithms

8. Design algorithms and data structures

- Visualize the process.
- Conceptualize the algorithm.
- Data representation is the essence of programming.
- Algorithms + Data structures = Programs

Algorithmic notation

- Natural language (English) — too ambiguous but informal.
- Programming language (even a high-level one) — too tedious but precise.
- Pseudocode: A blend of English and high-level programming language — informal and precise.
- Very-high level programming languages — Python, Haskell
- How about flowchart (flawchart)?

Algorithm	Program
Need not be formal (but unambiguous and precise)	Formal (hence, unambiguous and precise)
For design, correctness, analysis	Can be executed by a machine
Must terminate	Need not terminate

9. Design techniques

- Algorithm design techniques or strategies or tools — general approaches to common to a variety of problems
- Classification

10. Proving correctness

> Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.
>
> — Bertrand Meyer

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
>
> — E W Dijkstra

- A correct program need still be tested.
- Need to be simple to understand.

There are theories to reason about the correctness of programs.

Criticism:

- Most of the programs are not mission-critical!

- Learning curve of the theory is steep, and the benefit not commensurate with the effort.
- Real programs too large to calculate! So why calculate “small” programs?

Benefit:

- The goal is not correctness for correctness’ sake.
- The goal of program construction is to satisfy the input-ouput relation. Concern for correctness leads the stepwise construction of program toward the goal. The end program is correct by construction.
- Scale of the artifact does not obviate the need for knowing the fundamental tools of the trade!

Loop invariant and progress measure

Mathematical induction on iterative algorithms

- Linear search: search for a target in an array.
- Polish National Flag: partition an array into two parts.

Search for a target in an array, Linear search

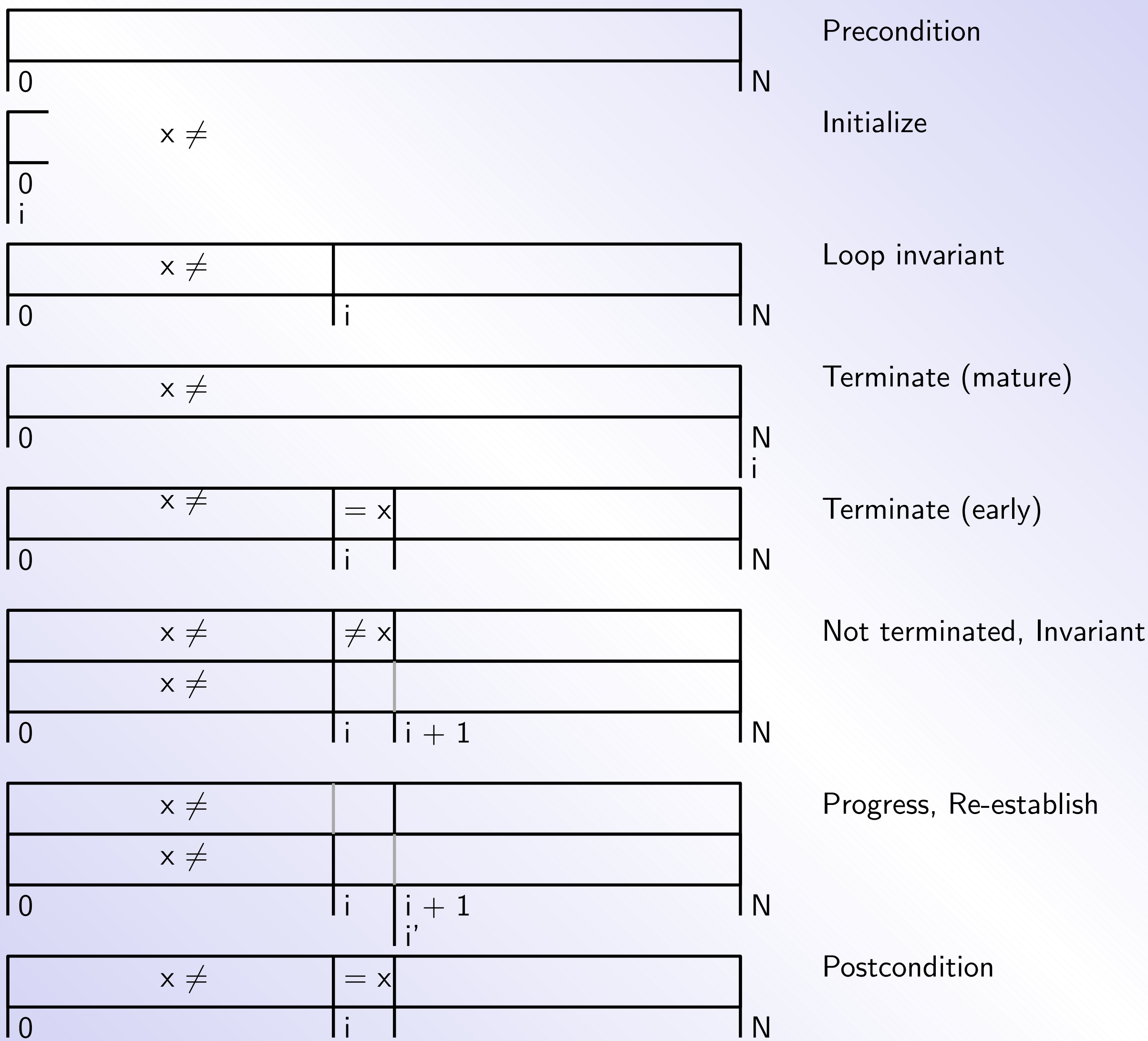
The input are an array `a` of N comparable items and a target item `x` to be searched in the array. The postcondition is: the output is the index of the target item in the array, if the target is present in the array; otherwise, the output is `N`, index of an invalid item.

A snapshot in the middle of the loop is that the subarray `a[0:i]` is processed (or conversely subarray `a[i:N]` is unprocessed). The basic iterative step is to compare the next item `a[i]` with the target `x`. In an iterative step, the processed subarray `a[0:i]` grows (unprocessed subarray `a[i:N]` shrinks).

Input: `a = [2, 5, 6, 8, 9, 10], N = 6, x = 9`

iteration	i	a	x
1	0	2, 5, 6, 8, 9, 10	9
2	1	2, 5, 6, 8, 9, 10	9
3	2	2, 5, 6, 8, 9, 10	9
4	3	2, 5, 6, 8, 9, 10	9
5	4	2, 5, 6, 8, 9, 10	9

Output: `i = 4`



Precondition
0
N
Initialize
x ≠
0
i
Loop invariant
x ≠
0
i
N
Terminate (mature)
x ≠
0
N
i
Terminate (early)
x ≠
= x
0
i
N
Not terminated, Invariant
x ≠
≠ x
x ≠
0
i
i + 1
N
Progress, Re-establish
x ≠
x ≠
0
i
i + 1
i'
N
Postcondition
x ≠
= x
0
i
N

Algorithm: LinearSearch (a, x)

input : An array a = [a_0, a_1, ..., a_{N-1}] of size N

output: Index i such that x $\neq$ [0:i] and a[i] = x or i = N

```
i ← 0
until i = N or a[i] = x do
    i ← i + 1
end
return i
```

Algorithm: LinearSearch (a, x)

input : An array a = [a_0, a_1, ..., a_{N-1}] of size N

output: Index i such that x ≠ [0:i] and a[i] = x or i = N

```
i ← 0
-- i = 0, x ∉ [0:i] ≡ x ∉ []
until i = N or a[i] = x do
    -- i ≠ N, x ∉ [0:i], x ≠ [i]
    -- x ∉ [0:i+1]
    i ← i + 1
    -- x ∉ [0:i']
end
-- x ∉ [0:i], x = [i]
-- x ∉ [0:i], i = N ≡ x ∉ [0:N]
return i
```

Partition an array into two parts, Polish National Flag

The input is an array a of `N` comparable items. The items are of two kinds, say Red and White. The postcondition is that the array is partitioned into two subarrays and the items so rearranged that

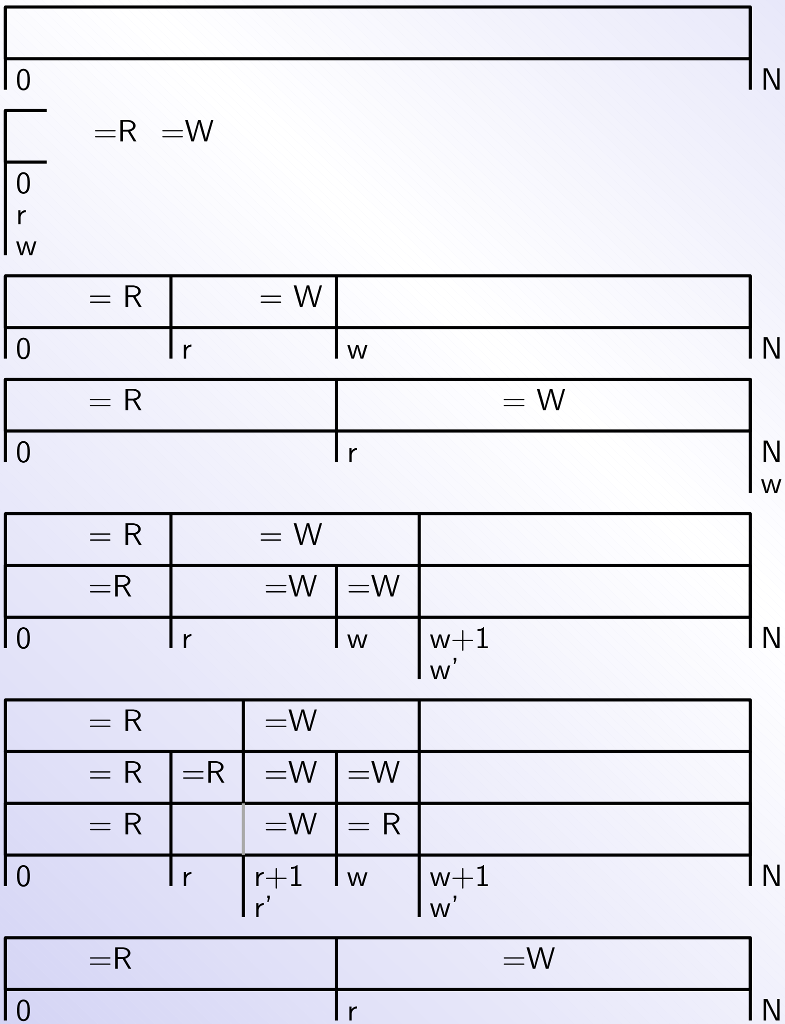
- subarray `[0:r]` has items of one kind (red), and
- subarray `[r:N]` has items of the other kind (white)

The only operations permitted are

- compare two items
- swap two items

Computer Science is no more about computers than astronomy is about telescopes or biology is about microscopes (E W Dijkstra).

Dutch National Flag



Precondition `r, w = 0, 0`

Initialize

Loop invariant

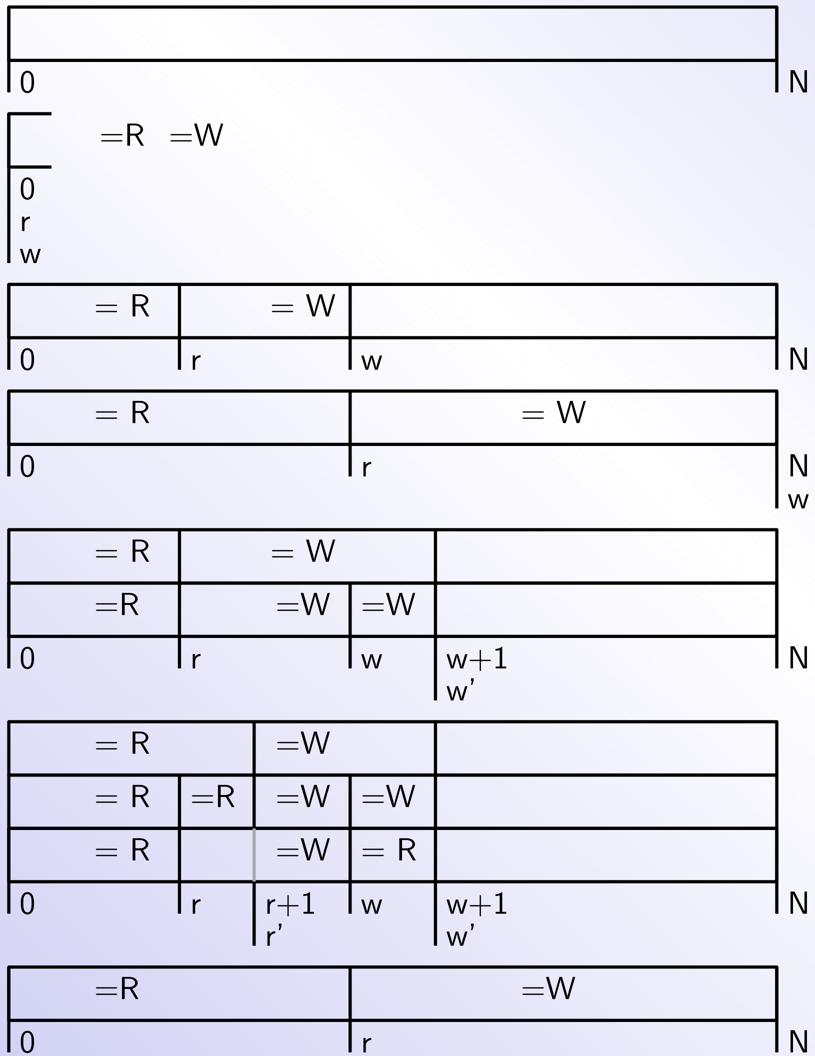
Terminate `w == N`

Progress, Re-establish `[w] == W | w' = w+1`

Progress, Re-establish

```
[w] == R |
    swap [r] [w]
    r, w = r+1, w+1
```

Postcondition



Precondition `r, w = 0, 0`

Initialize

Loop invariant

Terminate `w == N`

Progress, Re-establish `[w] == W | w' = w+1`

Progress, Re-establish

```
[w] == R |
    swap [r] [w]
    r, w = r+1, w+1
```

Postcondition

Algorithm: Partition (a)

input : A array a[0:N] of red and white items

output: Index `r` such that a[0:r] is red, a[r:N] is white

```
r, w ← 0, 0
until w = N do
    if a[w] = W then
        w ← w+1
    else
        swap (A, r, w)
        r, w ← r+1, w+1
    end
end
return r
```

Algorithm: Partition (a)

input : A array a[0:N] of red and white items

output: Index r such that a[0:r] is red, a[r:N] is white

```
r, w ← 0, 0
-- [] is red, [] is white
until w = N do
    -- [0:r] is red, [r:w] is white
    if a[w] = W then
        w ← w+1
        -- [0:r] is red, [r:w'] is white
    else
        swap (A, r, w)
        r, w ← r+1, w+1
        -- [0:r'] is red, [r':w'] is white
    end
end
return r
```

11. Analyzing algorithms for efficiency

- Time efficiency — how fast
- Space efficiency — how much memory

12. Syllabus

Unit I 9

Algorithm Analysis: Time Space Trade-off – Asymptotic Notations – Conditional asymptotic notation – Removing condition from the conditional asymptotic notation – Properties of big-Oh notation – Recurrence equations – Solving recurrence equations – Analysis of linear search.

Unit II 9

Divide and Conquer: General Method – Binary Search – Finding Maximum and Minimum – Merge Sort. Greedy Algorithms: General Method – Container Loading Knapsack Problem.

Unit III 9

Dynamic Programming: General Method – Multistage Graphs – All-Pair shortest paths – Optimal binary search trees – 0/1 Knapsack – Traveling salesperson problem.

Unit IV 9

Backtracking: General Method – 8 Queens problem – sum of subsets – graph coloring – Hamiltonian problem – knapsack problem.

Unit V 9

Graph Traversals: Connected Components – Spanning Trees – Biconnected components. Branch and Bound: General Methods (FIFO & LC) – 0/1 Knapsack problem. Introduction to NP-Hard and NP-Completeness.

Text Books

- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, "Computer Algorithms/C++", Second Edition, Universities Press, 2007. (For Units II to V)
- K.S. Easwarakumar, "Object Oriented Data Structures using C++", Vikas Publishing House pvt. Ltd., 2000 (For Unit I)

References

- T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2003.
- Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Pearson Education, 1999.