

Programming Concepts

Part 3: Where do algorithms come from?

So far:

- How to give a specification for an algorithmic problem
- A language for writing down algorithms
(= solutions to algorithmic problems)

Now: how do you come up with those solutions?

How to solve a problem

Polya: *How to solve it*. Princeton U.P. 1973

- (1) Understand the problem
- (2) Devise a plan for solving the problem
- (3) Carry out the plan
- (4) Evaluate the solution for accuracy

How to solve a problem

Polya: *How to solve it*. Princeton U.P. 1973

- (1) Understand the problem
- (2) Devise a plan for solving the problem
- (3) Carry out the plan
- (4) Evaluate the solution for accuracy

Steps **not sequential**

- (1) always comes first
- during (3) - may realise (1) not achieved
- during (3) may realise (2) is incorrect

Developing an algorithm

- (1) Understand the problem
- (2) Get an idea of how an algorithm might solve problem
- (3) Check out idea
- (4) Formulate idea as an algorithm
- (5) Evaluate algorithm for correctness, efficiency
- (6) Implement algorithm in favourite programming language

Developing an algorithm

- (1) Understand the problem
- (2) Get an idea of how an algorithm might solve problem
- (3) Check out idea
- (4) Formulate idea as an algorithm
- (5) Evaluate algorithm for correctness, efficiency
- (6) Implement algorithm in favourite programming language

Caveat:

There are no good algorithms for discovering algorithms

1. Understand the problem

- check problem specification
- does it make sense ?
- boundary cases ?
- tackle simple instances

Developing an algorithm

1. Understand the problem

- check problem specification
- does it make sense ?
- boundary cases ?
- tackle simple instances

Input: array A and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

1. Understand the problem

- check problem specification
- does it make sense ?
- boundary cases ?
- tackle simple instances

2. Get an idea

- Where from ?
- there are no good algorithms for getting ideas

Algorithm development

1. Understand the problem

- check problem specification
- does it make sense ?
- boundary cases ?
- tackle simple instances

2. Get an idea

- Where from ?
- there are no good algorithms for getting ideas

Input: array $A[1, \dots, n]$ and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

Idea: scan through the list until we find M

1. Understand the problem
 - check problem specification
 - does it make sense ?
 - boundary cases ?
 - tackle simple instances
2. Get an idea
 - Where from ?
 - there are no good algorithms for getting ideas
3. Two general approaches
 - top-down
 - bottom-up

Top-down problem solving

- (1) Don't solve the entire problem
- (2) Reduce problem to sub-problems
- (3) Tackle sub-problems
- (4) Eventually end up with trivial problems

Top-down problem solving

- (1) Don't solve the entire problem
- (2) Reduce problem to sub-problems
- (3) Tackle sub-problems
- (4) Eventually end up with trivial problems

Caveat:

Not necessarily sequential

- Reduction to sub-problems may not work
- Sub-problems may be more difficult than original problem

Bottom-up problem solving

1. Think directly about the problem itself
2. Examine simple instances
3. Solve simple/trivial instances of problem
4. Research algorithmic solutions to similar problems in literature
5. Try to understand intuitive mechanisms used in (3)
6. Try to extend (3) to slightly more complicated instances
7. Get an idea for general algorithm
8. Formulate potential algorithm, and evaluate

Summing up: Algorithm development

Best approach:

Use

- Bottom-up
- Top-down

approaches simultaneously

Summing up: Algorithm development

Best approach:

Use

- Bottom-up
- Top-down

approaches simultaneously

- There is no algorithm for ...
- Develop intuition from experience
- Develop familiarity with sources for common algorithms

Worked example 1: Searching an array

Searching an array

Problem specification:

Input: array $A[1 \dots n]$ and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

Searching an array

Problem specification:

Input: array $A[1 \dots n]$ and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

Get an idea from somewhere?

Searching an array

Problem specification:

Input: array $A[1 \dots n]$ and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

Get an idea from somewhere?

Straightforward:

- scan through the array A
- compare each element in turn with M

Searching an array

Problem specification:

Input: array $A[1 \dots n]$ and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

Get an idea from somewhere?

Straightforward:

- scan through the array A
- compare each element in turn with M

Write up algorithm

Searching an array

Problem specification:

Input: array $A[1 \dots n]$ and item M

Required output:

- true – if M occurs in A
- false – if M does not occur in A

Get an idea from somewhere?

Straightforward:

- scan through the array A
- compare each element in turn with M

Write up algorithm

- For-loop or While-loop ?

Sequential search: For-loop

Input: Array $A[1 \dots n]$, item M

Output: true, if M in A , false otherwise

FOUND \leftarrow *false*

for $i \leftarrow 1$ **to** n **do**

if $A[i] = M$ **then**
 FOUND \leftarrow *true*

return FOUND

Sequential search: For-loop

Input: Array $A[1 \dots n]$, item M

Output: true, if M in A , false otherwise

FOUND \leftarrow *false*

for $i \leftarrow 1$ **to** n **do**

if $A[i] = M$ **then**
 FOUND \leftarrow *true*

return FOUND

- Many unnecessary comparisons
- On average the entire array need not be searched

Sequential search: While-loop

Input: Array $A[1 \dots n]$, item M

Output: true, if M in A , false otherwise

FOUND \leftarrow *false*

PTR \leftarrow 1

while *not* FOUND *and* PTR $\leq n$ **do**

if $A[\text{PTR}] = M$ **then**

 FOUND \leftarrow *true*

 PTR \leftarrow PTR + 1

return FOUND

Worked example 2: Sorting an array

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Understand the problem:

Only interchanges within A allowed

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Understand the problem:

Only interchanges within A allowed

Do simple instances:

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Understand the problem:

Only interchanges within A allowed

Do simple instances:

C D A B

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Understand the problem:

Only interchanges within A allowed

Do simple instances:

C D A B \rightarrow A B C D

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Understand the problem:

Only interchanges within A allowed

Do simple instances:

C	D	A	B	→	A	B	C	D
M	A	G	F					

Sorting an array

Problem specification:

Input: array $A[1 \dots n]$

Required output: A sorted

Constraint: Sorting to be in situ

Understand the problem:

Only interchanges within A allowed

Do simple instances:

C D A B \rightarrow A B C D

M A G F \rightarrow A F G M

Sorting: we need an idea

Isolate effective interchanges

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element
- third most effective interchange:
Interchange $A[3]$ with third smallest element

Sorting: we need an idea

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element
- third most effective interchange:
Interchange $A[3]$ with third smallest element
fourth

Evaluate idea:

Sorting: we need an idea

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element
- third most effective interchange:
Interchange $A[3]$ with third smallest element
fourth

Evaluate idea:

D C A B

Sorting: we need an idea

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element
- third most effective interchange:
Interchange $A[3]$ with third smallest element
fourth

Evaluate idea:

D C A B \rightarrow A C D B

Sorting: we need an idea

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element
- third most effective interchange:
Interchange $A[3]$ with third smallest element
fourth

Evaluate idea:

D	C	A	B	→	A	C	D	B
A	C	D	B	→	A	B	D	C

Sorting: we need an idea

Isolate effective interchanges

- first most effective interchange:
Interchange $A[1]$ with smallest element
- second most effective interchange:
Interchange $A[2]$ with second smallest element
- third most effective interchange:
Interchange $A[3]$ with third smallest element
fourth

Evaluate idea:

D	C	A	B	→	A	C	D	B
A	C	D	B	→	A	B	D	C
A	B	D	C	→	A	B	C	D

Sorting: a fuzzy “algorithm”

Roughly:

for $i \leftarrow 1$ **to** n **do**

$\text{MIN} \leftarrow$ index of i^{th} smallest element of $A[1 \dots n]$;
 $A[i] \leftrightarrow A[\text{MIN}]$

Sorting: a fuzzy “algorithm”

Roughly:

for $i \leftarrow 1$ **to** n **do**

 | $\text{MIN} \leftarrow$ index of i^{th} smallest element of $A[1 \dots n]$;
 | $A[i] \leftrightarrow A[\text{MIN}]$

Note:

- This is not an algorithm
- This is a reduction

Sorting: a fuzzy “algorithm”

Roughly:

for $i \leftarrow 1$ **to** n **do**

 | $\text{MIN} \leftarrow$ index of i^{th} smallest element of $A[1 \dots n]$;
 | $A[i] \leftrightarrow A[\text{MIN}]$

Note:

- This is not an algorithm
- This is a **reduction** to the problem:
 Find the i^{th} smallest element in an array

Progressing the algorithm development

We need to solve:

Find the i^{th} smallest element in an array

Progressing the algorithm development

We need to solve:

Find the i^{th} smallest element in an array

Or do we?

Progressing the algorithm development

We need to solve:

Find the i^{th} smallest element in an array

Or do we?

for $i \leftarrow 1$ **to** n **do**

$MIN \leftarrow$ index of i^{th} smallest element;

$A[i] \leftrightarrow A[MIN]$ // invariant: $A[1 \dots i]$ sorted

Progressing the algorithm development

We need to solve:

Find the i^{th} smallest element in an array

Or do we?

for $i \leftarrow 1$ **to** n **do**

$\text{MIN} \leftarrow$ index of i^{th} smallest element;

$A[i] \leftrightarrow A[\text{MIN}]$ // invariant: $A[1 \dots i]$ sorted

Because of invariant:

- i^{th} smallest element of $A[1 \dots n]$
- **smallest** element of $A[i \dots n]$

are the same

for $i \leftarrow 1$ **to** n **do**

$\text{MIN} \leftarrow$ index of smallest element of $A[i \dots n]$;

$A[i] \leftrightarrow A[\text{MIN}]$ // invariant: $A[1 \dots i]$ sorted

for $i \leftarrow 1$ **to** n **do**

$\text{MIN} \leftarrow$ index of smallest element of $A[i \dots n]$;

$A[i] \leftrightarrow A[\text{MIN}]$ // invariant: $A[1 \dots i]$ sorted

Progressing:

for $i \leftarrow 1$ **to** n **do**

$\text{MIN} \leftarrow$ index of smallest element of $A[i \dots n]$;

$A[i] \leftrightarrow A[\text{MIN}]$ // invariant: $A[1 \dots i]$ sorted

Progressing:

- Look up algorithm for finding smallest element in an array

for $i \leftarrow 1$ **to** n **do**

$\text{MIN} \leftarrow$ index of smallest element of $A[i \dots n]$;

$A[i] \leftrightarrow A[\text{MIN}]$ // invariant: $A[1 \dots i]$ sorted

Progressing:

- Look up algorithm for finding smallest element in an array
- Adapt to finding index of smallest element in $A[i \dots n]$

Proposed algorithm

Input: Array $A[1 \dots n]$

Output: array A sorted

for $i \leftarrow 1$ **to** n **do**

 // maintains $A[1 \dots i]$ sorted

 MINPTR $\leftarrow i$

for $j \leftarrow (i + 1)$ **to** n **do**

if $A[j] < A[\text{MINPTR}]$ **then**

 MINPTR $\leftarrow j$

$A[i] \leftrightarrow A[\text{MINPTR}]$

return A

Proposed algorithm

Input: Array $A[1 \dots n]$

Output: array A sorted

for $i \leftarrow 1$ **to** n **do**

 // maintains $A[1 \dots i]$ sorted

 MINPTR $\leftarrow i$

for $j \leftarrow (i + 1)$ **to** n **do**

if $A[j] < A[\text{MINPTR}]$ **then**

 MINPTR $\leftarrow j$

$A[i] \leftrightarrow A[\text{MINPTR}]$

return A

Evaluate algorithm for correctness, efficiency

Proposed algorithm

Input: Array $A[1 \dots n]$

Output: array A sorted

for $i \leftarrow 1$ **to** n **do**

 // maintains $A[1 \dots i]$ sorted

 MINPTR $\leftarrow i$

for $j \leftarrow (i + 1)$ **to** n **do**

if $A[j] < A[\text{MINPTR}]$ **then**

 MINPTR $\leftarrow j$

$A[i] \leftrightarrow A[\text{MINPTR}]$

return A

Evaluate algorithm for correctness, efficiency

Last iteration of loop unnecessary

Selection sort

Input: Array $A[1 \dots n]$

Output: Array A sorted

for $i \leftarrow 1$ **to** $(n - 1)$ **do**

 // maintains $A[1 \dots i]$ sorted

 MINPTR $\leftarrow i$

for $j \leftarrow (i + 1)$ **to** n **do**

 // finds smallest item in $A[i \dots n]$

if $A[j] < A[\text{MINPTR}]$ **then**

 MINPTR $\leftarrow j$

$A[i] \leftrightarrow A[\text{MINPTR}]$ // swap items

return A

How did we get here?

- We iterated on our first idea
- Key point: just need to make sure $A[1, \dots, i]$ is sorted at every point (understand the problem!)
- This **invariant** made it easier to design the algorithm, and helps us see why it's correct

Sorting arrays: another idea

$A[1] \ A[2] \ \dots \ A[j] \ A[j+1] \ \dots \ A[n]$

is not sorted because some $A[j]$ is greater than $A[j+1]$

Sorting arrays: another idea

$A[1] \ A[2] \ \dots \ A[j] \ A[j+1] \ \dots \ A[n]$

is not sorted because some $A[j]$ is greater than $A[j+1]$

Solution:

Scan through the array switching all offending contiguous pairs

Sorting arrays: another idea

$A[1] \ A[2] \ \dots \ A[j] \ A[j+1] \ \dots \ A[n]$

is not sorted because some $A[j]$ is greater than $A[j+1]$

Solution:

Scan through the array switching all offending contiguous pairs

Pseudo-code:

```
for  $j \leftarrow 1$  to  $n - 1$  do  
  | if  $A[j] > A[j + 1]$  then  $A[j] \leftrightarrow A[j + 1]$ 
```

Sorting arrays: another idea

$A[1] \ A[2] \ \dots \ A[j] \ A[j+1] \ \dots \ A[n]$

is not sorted because some $A[j]$ is greater than $A[j+1]$

Solution:

Scan through the array switching all offending contiguous pairs

Pseudo-code:

```
for  $j \leftarrow 1$  to  $n - 1$  do  
  | if  $A[j] > A[j + 1]$  then  $A[j] \leftrightarrow A[j + 1]$ 
```

Evaluate the idea

Evaluate idea

- 5 A
- 4 G
- 3 B
- 2 D
- 1 E

Evaluate idea

5	A	A
4	G	G
3	B	B
2	D	D
1	E	E

?) ?

Evaluate idea

5	A	A	A
4	G	G	G
3	B	B	B
2	D	D	E
1	E	E	D

Diagram illustrating a sequence of letters (A, G, B, D, E) arranged in three columns. Red annotations include a bracket on the right side of the third column (rows 2-3) and a bracket on the right side of the second column (rows 2-3), both with a question mark.

Evaluate idea

5	A	A	A	A
4	G	G	G	G
3	B	B	B	E
2	D	D	E	B
1	E	E	D	D

Diagram illustrating a sequence of letters (A, G, B, D, E) arranged in a grid. Red curved lines and question marks indicate a sequence of operations or a path through the letters:

- Red line from G (row 4, column 4) to E (row 3, column 4) with a question mark.
- Red line from B (row 3, column 3) to E (row 2, column 3) with a question mark.
- Red line from D (row 2, column 2) to E (row 1, column 2) with a question mark.

Evaluate idea

5	A	A	A	A	A
4	G	G	G	G	G
3	B	B	B	E	E
2	D	D	E	B	B
1	E	E	D	D	D

Diagram illustrating a sequence of letters (A, G, B, E, D) arranged in columns, with red curved arrows indicating transitions between letters in adjacent columns. The arrows are labeled with question marks, suggesting a sequence of transitions or a path to be evaluated.

- Column 1: A, G, B, D, E
- Column 2: A, G, B, D, E
- Column 3: A, G, B, E, D
- Column 4: A, G, E, B, D
- Column 5: A, G, E, B, D

Red curved arrows and question marks indicate transitions between letters in adjacent columns:

- From Column 1 to Column 2: A to A, G to G, B to B, D to D, E to E.
- From Column 2 to Column 3: A to A, G to G, B to B, E to E, D to D.
- From Column 3 to Column 4: A to A, G to G, E to E, B to B, D to D.
- From Column 4 to Column 5: A to A, G to G, E to E, B to B, D to D.

Evaluate idea

5	A	A	A	A	A	G
4	G	G	G	G	G	A
3	B	B	B	E	E	E
2	D	D	E	B	B	B
1	E	E	D	D	D	D

Diagram illustrating a sequence of letters (A, G, B, D, E) arranged in columns, with red curved arrows indicating a shift or transformation between adjacent columns. The arrows suggest a shift of one position down for each column, with question marks indicating uncertainty or evaluation of the idea.

Evaluate idea

5	A	A	A	A	A	G
4	G	G	G	G	G	A
3	B	B	B	E	E	E
2	D	D	E	B	B	B
1	E	E	D	D	D	D

Diagram illustrating a sorting process (likely bubble sort) on an array of letters. Red curved arrows indicate comparisons and swaps between adjacent elements. Question marks indicate elements being compared or the state after a swap.

- Row 5: A, A, A, A, A, G. A red arrow points from the 5th A to the G.
- Row 4: G, G, G, G, G, A. A red arrow points from the 5th G to the A.
- Row 3: B, B, B, E, E, E. A red arrow points from the 3rd B to the 4th E.
- Row 2: D, D, E, B, B, B. A red arrow points from the 2nd D to the 3rd E.
- Row 1: E, E, D, D, D, D. A red arrow points from the 1st E to the 2nd E.

- Final array not sorted

Evaluate idea

5	A	A	A	A	A	G
4	G	G	G	G	G	A
3	B	B	B	E	E	E
2	D	D	E	B	B	B
1	E	E	D	D	D	D

- Final array not sorted
- Try more examples; understand what is going on

Evaluate idea

5	A	A	A	A	A	G
4	G	G	G	G	G	A
3	B	B	B	E	E	E
2	D	D	E	B	B	B
1	E	E	D	D	D	D

Diagram illustrating the state of an array during a sorting process (likely Bubble Sort). The array is represented by rows (indices 1 to 5) and columns (elements). Red curved arrows and question marks indicate comparisons and potential swaps between adjacent elements in each row.

- Final array not sorted
- Try more examples; understand what is going on
- Largest item always bubbles to top

Evaluate idea

5	A	A	A	A	A	G
4	G	G	G	G	G	A
3	B	B	B	E	E	E
2	D	D	E	B	B	B
1	E	E	D	D	D	D

Diagram illustrating a sorting process (likely bubble sort) on an array of letters. The array is shown in 6 columns and 5 rows. Red curved arrows indicate comparisons and swaps between adjacent elements in each row. Question marks indicate the current state of the array after a swap or comparison.

- Final array not sorted
- Try more examples; understand what is going on
- Largest item always bubbles to top
- New idea: repeat this procedure again and again, so larger items gradually bubble upwards

Evaluate idea

5	A	A	A	A	A	?	G
4	G	G	G	G	G	?	A
3	B	B	B	E	E		E
2	D	D	E	B	B		B
1	E	E	D	D	D		D

- Final array not sorted
- Try more examples; understand what is going on
- Largest item always bubbles to top
- New idea: repeat this procedure again and again, so larger items gradually bubble upwards
- Repeat $n - 1$ times where n size of array

Proposed algorithm

Input: Array $A[1 \dots n]$

Output: Array A sorted

for $i \leftarrow 1$ **to** $(n - 1)$ **do**

for $j \leftarrow 1$ **to** $(n - 1)$ **do**

if $A[j] > A[j + 1]$ **then** $A[j] \leftrightarrow A[j + 1]$

Proposed algorithm

Input: Array $A[1 \dots n]$

Output: Array A sorted

for $i \leftarrow 1$ **to** $(n - 1)$ **do**

for $j \leftarrow 1$ **to** $(n - 1)$ **do**

if $A[j] > A[j + 1]$ **then** $A[j] \leftrightarrow A[j + 1]$

Evaluate the proposal

- Run on examples

Proposed algorithm

Input: Array $A[1 \dots n]$

Output: Array A sorted

for $i \leftarrow 1$ **to** $(n - 1)$ **do**

for $j \leftarrow 1$ **to** $(n - 1)$ **do**
 if $A[j] > A[j + 1]$ **then** $A[j] \leftrightarrow A[j + 1]$

Evaluate the proposal

- Run on examples
- later iterations of inner loop unnecessary

Bubblesort

Input: Array $A[1 \dots n]$

Output: Array A sorted

for $i \leftarrow 1$ **to** $(n - 1)$ **do**

 // maintains $A[(n - i + 1) \dots n]$ sorted

for $j \leftarrow 1$ **to** $(n - 1)$ **do**

 // bubbles up largest element in

$A[1 \dots (n - i + 1)]$

if $A[j] > A[j + 1]$ **then**

$A[j] \leftrightarrow A[j + 1]$

Summary

- Algorithmic problem solving: understand the problem!
- Decomposing the problem into smaller problems.
- Searching.
- Sorting.
- Can we do better for each of the above?

To Do:

- Exercises and Homeworks: check the solutions so far.
- Homeworks: Something to do for self-study.
- Exercises: Will be discussed in the exercise class.

Ask any questions at the Helpdesk/Exercise sessions.