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

Developing an algorithm

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

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

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 $A[1, \ldots, n]$ 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

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
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 $A[1 \ldots n]$ 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 $A[1 \ldots n]$ 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 $A[1 \ldots n]$ 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 $A[1 \ldots n]$, item M

Output: true, if M in A, false otherwise

$\textsc{found} \leftarrow \textit{false}$

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

    if $A[i] = M$ **then**

        $\textsc{found} \leftarrow \textit{true}$

return $\textsc{found}$

Sequential search: For-loop

Input: Array $A[1 \ldots n]$, item M
Output: true, if M in A, false otherwise

$\textsc{found} \leftarrow \textit{false}$
for $i \leftarrow 1$ **to** n **do**
  if $A[i] = M$ **then**
    $\textsc{found} \leftarrow \textit{true}$
return $\textsc{found}$

- 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

$\textsc{found} \leftarrow \textit{false}$

$\textsc{ptr} \leftarrow 1$

while *not* $\textsc{found}$ *and* $\textsc{ptr} \leq n$ **do**

    if $A[\textsc{ptr}] = M$ **then**

        $\textsc{found} \leftarrow \textit{true}$

    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

return $\textsc{found}$

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 $A[1 \ldots n]$

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 $A[1 \ldots n]$

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 $A[1 \ldots n]$

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

Sorting an array

Problem specification:

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

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

Sorting: we need an idea

Isolate effective interchanges

- first most effective interchange:

  Interchange $A[1]$ with 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

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

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	$\rightarrow$	A	C	D	B
A	C	D	B	$\rightarrow$	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 $\rightarrow$ A C D B

A C D B $\rightarrow$ A B D C

A B D C $\rightarrow$ A B C D

Sorting: a fuzzy “algorithm”

Roughly:

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

  $\textsc{min} \leftarrow$ index of i^{th} smallest element of A[1...n];

  $A[i] \leftrightarrow A[\textsc{min}]$

Sorting: a fuzzy “algorithm”

Roughly:

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

  $\textsc{min} \leftarrow$ index of i^{th} smallest element of A[1...n];

  $A[i] \leftrightarrow A[\textsc{min}]$

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...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**
  $\textsc{min} \leftarrow$ index of i^{th} smallest element;
  $A[i] \leftrightarrow A[\textsc{min}]$ `// invariant:` $A[1 \ldots i]$ `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**

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

  $A[i] \leftrightarrow A[\textsc{min}]$ `// invariant:` $A[1 \ldots i]$ `sorted`

Because of invariant:

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

are the same

Revising the algorithm

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

  $\textsc{min} \leftarrow$ index of smallest element of $A[i \ldots n]$;

  $A[i] \leftrightarrow A[\textsc{min}]$ `// invariant:` $A[1 \ldots i]$ `sorted`

Revising the algorithm

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

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

  $A[i] \leftrightarrow A[\textsc{min}]$ `// invariant:` $A[1 \dots i]$ `sorted`

Progressing:

Revising the algorithm

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

  $\textsc{min} \leftarrow$ index of smallest element of $A[i \ldots n]$;

  $A[i] \leftrightarrow A[\textsc{min}]$ `// invariant:` $A[1 \ldots i]$ `sorted`

Progressing:

- Look up algorithm for finding smallest element in an array

Revising the algorithm

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

$\quad$ $\textsc{min} \leftarrow$ index of smallest element of $A[i \ldots n]$;

$\quad$ $A[i] \leftrightarrow A[\textsc{min}]$ `// invariant:` $A[1 \ldots i]$ `sorted`

Progressing:

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

Proposed algorithm

Input: Array $A[1 \dots n]$
Output: array A sorted

```
for i ← 1 to n do
    // maintains A[1...i] sorted
    MINPTR ← i
    for j ← (i + 1) to n do
        if A[j] < A[MINPTR] then
            MINPTR ← j
    A[i] ↔ A[MINPTR]
return A
```

Proposed algorithm

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

Output: array A sorted

```
for i ← 1 to n do
    // maintains A[1...i] sorted
    MINPTR ← i
    for j ← (i + 1) to n do
        if A[j] < A[MINPTR] then
            MINPTR ← j
    A[i] ↔ A[MINPTR]
return A
```

Evaluate algorithm for correctness, efficiency

Proposed algorithm

Input: Array $A[1 \ldots n]$
Output: array A sorted

```
for i ← 1 to n do
    // maintains A[1...i] sorted
    MINPTR ← i
    for j ← (i + 1) to n do
        if A[j] < A[MINPTR] then
            MINPTR ← j
    A[i] ↔ A[MINPTR]
return A
```

Evaluate algorithm for correctness, efficiency

Last iteration of loop unnecessary

Input: Array $A[1 \ldots n]$
Output: Array A sorted
for $i \leftarrow 1$ **to** $(n-1)$ **do**
    `// maintains` $A[1 \ldots i]$ `sorted`
    $\text{MINPTR} \leftarrow i$
    for $j \leftarrow (i+1)$ **to** n **do**
        `// finds smallest item in` $A[i \ldots n]$
        if $A[j] < A[\text{MINPTR}]$ **then**
            $\text{MINPTR} \leftarrow j$
    $A[i] \leftrightarrow A[\text{MINPTR}]$ `// swap items`
return A

Stepping back

How did we get here?

- We iterated on our first idea
- Key point: just need to make sure $A[1, \ldots, i]$ 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] . . . . . . A[j] A[j+1] . . . . . . A[n]

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

Sorting arrays: another idea

A[1] A[2] ...... A[j] A[j+1] ...... 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] ...... A[j] A[j+1] ...... 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] ...... A[j] A[j+1] ...... 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	

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	

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	

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

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

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

- 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

- 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

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

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 ← 1 to (n − 1) do
    // maintains A[(n − i + 1) ... n] sorted
    for j ← 1 to (n − 1) do
        // bubbles up largest element in
            A[1 ... (n − i + 1)]
        if A[j] > A[j + 1] then
            A[j] ↔ 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.