

Course «Algorithms and Data Structures»

6B06102 – Information Systems
6B06109 – Artificial Intelligence
6B06103 – Computer Engineering
6B06104 – Computer Science

Lecture 8

Other Topics in Algorithms

Textbook: A. Levitin, Section 2.1, Chapters 7–9

Six questions for today

1. How much memory does an algorithm use?
2. Can extra memory buy us speed?
3. How can adding to a dynamic array be O(1)?
4. How can two indices replace two nested loops?
5. How do we prove that a loop is correct?
6. When is the obvious choice the wrong one?

None of these needs a new data structure. They’re ways of thinking.

Memory is a cost too

Space complexity: extra memory as a function of n, without the input itself.

Algorithm	Time	Extra memory	Why
MaxElement	O(n)	O(1)	one variable
FibLoop	O(n)	O(1)	the last two numbers
Factorial, recursive	O(n)	O(n)	a stack frame per call
Binary search, iterative	O(log n)	O(1)	l, r and m
Binary search, recursive	O(log n)	O(log n)	a stack frame per level

The call stack is real memory, and it counts.

Trading memory for time: prefix sums

Many questions: “what is the sum of A[l..r]?” A loop takes O(n) for each one.

```
ALGORITHM PrefixSums(A[0..n – 1])
    P[0] ← 0
    for i ← 0 to n – 1 do
        P[i + 1] ← P[i] + A[i]
    return P
```

$$A[l] + \cdots + A[r] = P[r+1] - P[l]$$

Prefix sums: an example

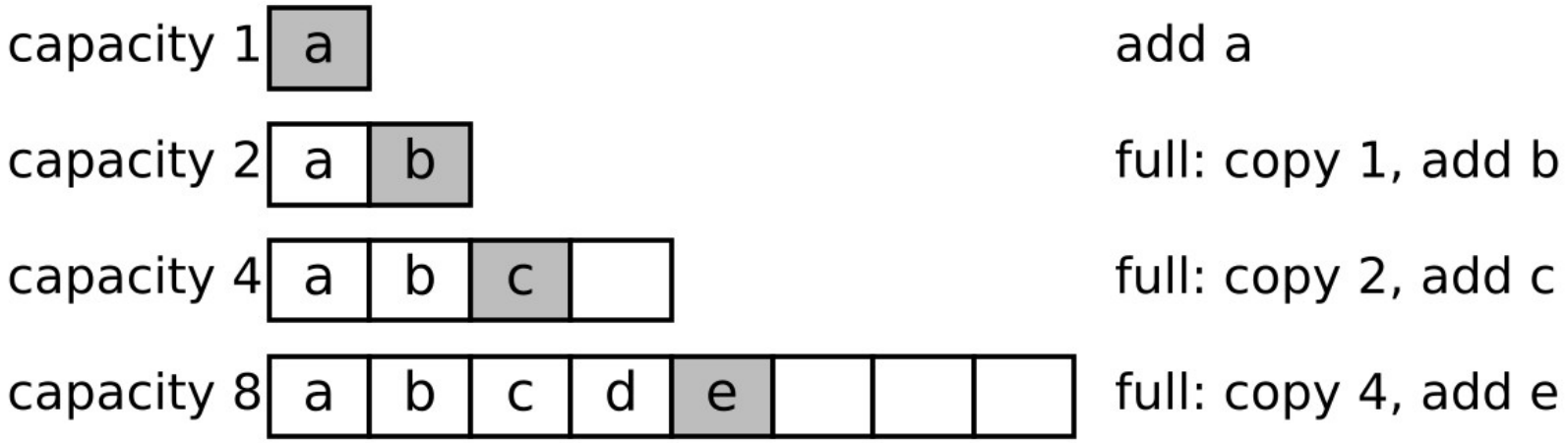
i	0	1	2	3	4	5	6	7	8
A[i]	3	1	4	1	5	9	2	6	
P[i]	0	3	4	8	9	14	23	25	31

$$A[2] + \cdots + A[5] = P[6] - P[2] = 23 - 4 = 19$$

- **O(n)** time once and **O(n)** extra memory
- Then every question takes **O(1)**

This is the time–memory trade-off. Hash tables (week 12) make the same deal.

A dynamic array that doubles



Copying is expensive, but it happens rarely. How rarely?

Counting the work: 16 additions

Addition	1	2	3	4	5	6–8	9	10–16
Copies	0	1	2	0	4	0	8	0

$$16 \text{ writes} + (1 + 2 + 4 + 8) \text{ copies} = 31 \text{ operations}$$

- Fewer than **2** operations per addition here
- In general, n additions take fewer than **3n** operations

Amortized cost

One addition

O(n) in the worst case, when the array must be copied

A sequence of additions

O(1) amortized: the average over the whole sequence

- Not the average case: no lucky input is needed
- Growing by a fixed 10 cells would cost **$O(n^2)$**: doubling is what makes it work

Two pointers: the idea

A sorted array and a number T. Are there two elements that add up to T?

```
ALGORITHM TwoSum(A[0..n – 1], T)
    l ← 0; r ← n – 1
    while l < r do
        s ← A[l] + A[r]
        if s = T return (l, r)
        else if s < T  l ← l + 1
        else           r ← r – 1
    return "no such pair"
```

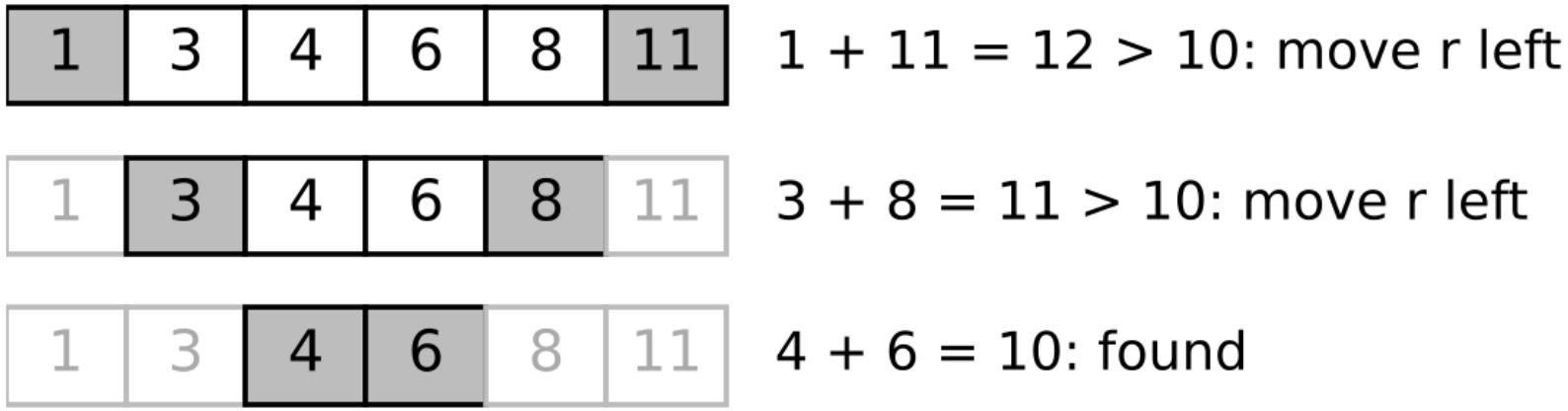
Too large? Move r left. Too small? Move l right.

Tracing by hand: T = 10

Step	l	r	A[l]	A[r]	Sum	Decision
1	0	5	1	11	12	too large, r ← 4
2	0	4	1	8	9	too small, l ← 1
3	1	4	3	8	11	too large, r ← 3
4	1	3	3	6	9	too small, l ← 2
5	2	3	4	6	10	found

The array: 1, 3, 4, 6, 8, 11

Why it can’t miss a pair



- If the sum is too large, A[r] can’t be part of any answer: throw it away
- **O(n)** time and **O(1)** extra memory, not O(n^2)

Sorted data lets us throw candidates away without looking at them.

Is the loop correct? Loop invariants

A **loop invariant** is a statement that is true before every iteration.

Start

It’s true before the first iteration.

Step

If it’s true before an iteration, it’s true after it.

End

When the loop stops, it gives us the answer.

Testing can show errors, but not their absence. An invariant proves the loop for every input.

Example: MaxElement

```
ALGORITHM MaxElement(A[0..n – 1])
    maxval ← A[0]
    for i ← 1 to n – 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

Invariant: before iteration i, maxval is the largest of A[0..i – 1]

- **Start:** for i = 1, A[0..0] has one element, maxval = A[0]
- **Step:** either A[i] replaces maxval or it doesn’t; maxval is the largest of A[0..i]
- **End:** at i = n, maxval is the largest of the whole array

The greedy idea

At each step, take the choice that looks best right now. Never go back.

```
ALGORITHM GreedyChange(amount, coins)
    //coins are sorted from largest to smallest
    for each coin c in coins do
        while amount ≥ c do
            give one coin c
            amount ← amount – c
```

When greedy fails

Coins 25, 10, 5, 1; amount 30

Greedy: 25 + 5, two coins. The best possible.

Coins 4, 3, 1; amount 6

Greedy: 4 + 1 + 1, three coins. But 3 + 3 needs only two!

Prove that a greedy idea works, or find a counterexample. One counterexample is enough.

Remember the answers: memoization

```
ALGORITHM FibMemo(n)
    //memo[0..n] is filled with −1 before the first call
    if n ≤ 1 return n
    if memo[n] ≠ −1 return memo[n]
    memo[n] ← FibMemo(n − 1) + FibMemo(n − 2)
    return memo[n]
```

	Plain recursion	**With memoization**
Calls for n = 10	177	19
Time	exponential	O(n)

The main idea of dynamic programming. You'll study it in later courses.

Common mistakes

- Forgetting that recursion uses memory
- Saying a dynamic array adds in O(1) in the worst case: it’s O(1) **amortized**
- Using two pointers on an array that isn’t sorted
- Trusting a greedy algorithm because it worked on one example
- Using memo[n] without checking that it has been filled

Summary

- Memory is a cost too, and the call stack counts.
- Extra memory can buy speed: prefix sums, memoization.
- A dynamic array that doubles adds in O(1) amortized.
- Sorted data lets two pointers run in O(n), not O(n^2).
- An invariant proves a loop; a counterexample rejects a greedy idea.

Review questions

1. For A = 2, 7, 1, 8, 2, 8, build the prefix sums P and use them to find the sum of A[1..4].
2. Why is adding to a dynamic array O(1) amortized if one addition can take O(n) steps? What would change if the array grew by 10 cells each time?
3. For coins 5, 4, 1, name an amount for which GreedyChange doesn’t use the fewest coins.

Reading: Levitin, Section 2.1, Chapters 7–9; Cormen et al., Section 2.1, Chapters 14–16.