Recursion: The Beginning

Yufei Tao

ITEE
University of Queensland

This lecture is the inception of a powerful technique called recursion.

If used judiciously, this technique can simplify the design of an algorithm significantly, by guiding us to see everything from a new, elegant, and neat perspective. Often times, it leads to a simpler running-time analysis as well!

To make our point, we will first "re-discover" the binary search algorithm, but in a remarkable manner by way of recursion. Then, we will use the new technique to attack a new problem: sorting.

Recall:

The Dictionary Search Problem

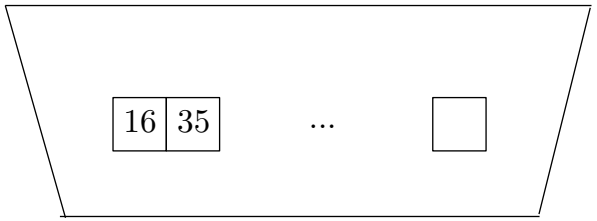
Problem Input:

In the memory, a set S of n integers has been arranged in ascending order at the memory cells from address 1 to n. The value of n has been placed in Register 1 of the CPU. Another integer v has been placed in Register 2 of the CPU.

Goal:

Design an algorithm to determine whether v exists in S.

A "yes"-input with $n = 16$

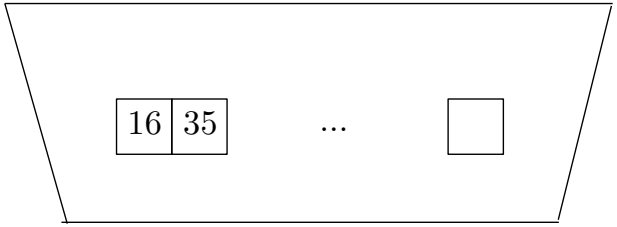


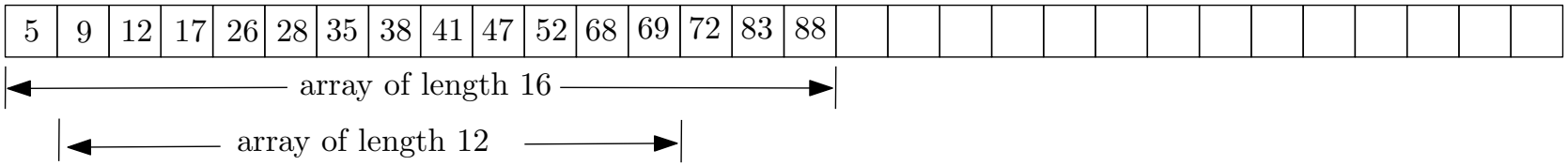
5	9	12	17	26	28	35	38	41	47	52	68	69	72	83	88														

Array

An **array** of **length** n is a sequence of n elements such that

- they are stored consecutively in memory (i.e., the first element is immediately followed by the second, and then by the third, and so on);
- every element occupies the same number of memory cells.





With the concept of array, we now redefine the dictionary search problem:

The Dictionary Search Problem (Redefined)

Problem Input:

A set S of n integers has been arranged in ascending order in an array of length n. You are given the value of n and another integer v inside the CPU.

Goal:

Design an algorithm to determine whether v exists in S.

Recursion in General

The idea of recursion is to carry out two steps:

1. **[Base Case]**
   Solve the case where the problem size $n = 1$ and 0 (usually trivial).
2. **[Inductive Case]**
   Solve the problem with a problem size $n > 1$ by reducing n.

Binary Search (Re-discovered)

Base Case ($n = 0$ or 1)

Trivial:

- If $n = 0$, then simply return "no".
- If $n = 1$, compare v to the (only) element in the array in $O(1)$ time (i.e., constant time without caring about what the constant is).

Binary Search (Re-discovered)

Inductive Case (arbitrary $n > 1$)

1. Compare v to the middle element e of the array. If $v = e$, return "yes" and done.
2. Otherwise:
   - 2.1 If $v < e$, solve the problem in the part of the array before e;
   - 2.2 If $v > e$, solve the problem in the part of the array after e.

Our algorithm description is now complete!

A Bottom-Up View of Recursion

Our algorithm essentially does the same work as our previous version of binary search (described using pesudo-code in a previous lecture).

This becomes logically clear if you think about it as follows. The base case settles the problem with $n = 1$. Now, let us focus on the problem with $n = 2$. At Step 2.1 or 2.2 (of the previous slide), we are facing another dictionary search problem, but with problem size 0 or 1! So, think no further—we already know how to solve it.

Now, focus on the problem with $n = 3$. At Step 2.1 or 2.2, we face a problem with size 1—again, we know how to solve it.

Focus on the problem with $n = 4$. After Step 2.1 or 2.2, we face a problem with size 2—done deal!

Etc.

Analysis of Binary Search

So it remains to solve the recurrence (c_1, c_2 are constants whose values we do not care):

$$\begin{aligned} f(1) &= c_1 \\ f(n) &\leq c_2 + f(n/2) \end{aligned}$$

An easy way of doing so is the expansion method, which simply expands $f(n)$ all the way down:

$$\begin{aligned} f(n) &\leq c_1 + f(n/2) \\ &\leq c_2 + c_2 + f(n/2^2) \\ &\leq c_2 + c_2 + c_2 + f(n/2^3) \\ &\leq \underbrace{c_2 + \ldots + c_2}_{\log_2 n \text{ of them}} + f(1) \\ &= c_2 \cdot \log_2 n + c_1 = O(\log n). \end{aligned}$$

Analysis of Binary Search

Technically speaking, our previous analysis holds only when n is a power of 2 (otherwise, some $n/2^i$ along the way is a non-integer, making $f(n/2^i)$ undefined).

We can account for this easily using a rounding approach. Suppose that n is not a power of 2. Let n' be the least power of 2 that is larger than n. It thus holds that $n' < 2n$ (otherwise, n' is not the least).

We then have:

$$\begin{aligned} f(n) &\leq f(n') \\ &\leq c_2 \cdot \log_2 n' + c_1 \text{ (proved earlier)} \\ &< c_2 \cdot \log_2(2n) + c_1 \\ &= c_2(1 + \log_2 n) + c_1 \\ &= c_2 \log_2 n + c_1 + c_2 = O(\log n). \end{aligned}$$

Next, we switch our attention to the sorting problem, which is a very classical problem in computer science, and is worth several lectures' discussion.

The Sorting Problem

Problem Input:

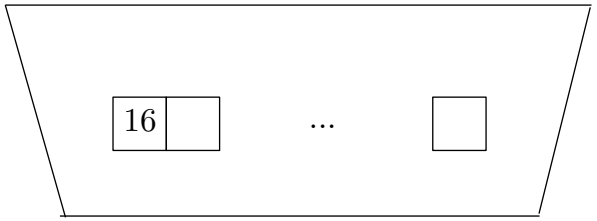
A set S of n integers is given in an array of length n. The value of n is inside the CPU (i.e., in a register).

Goal:

Design an algorithm to store S in an array where the elements have been arranged in ascending order.

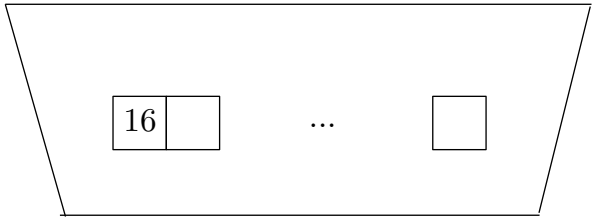
Example

Input:



38	28	88	17	26	41	72	83	69	47	12	68	5	52	35	9													

Output:



5	9	12	17	26	28	35	38	41	47	52	68	69	72	83	88													

We will use recursion to design our first sorting algorithm, called **selection sort**.

Selection Sort

Base Case ($n = 1$)

Trivial. Nothing to sort. Return directly.

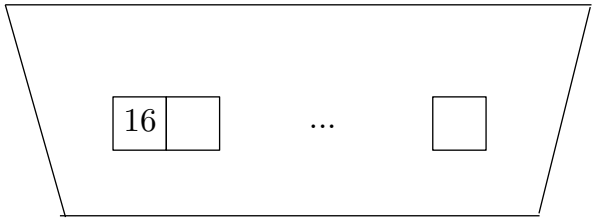
Selection Sort

Inductive Case (arbitrary $n > 1$)

1. Scan all the elements in the array to identify the largest one e_{max}.
2. Swap the positions of e_{max} and the last (i.e., n-th) element of the array (after which e_{max} is at the end of the array).
3. Sort the first $n - 1$ elements.

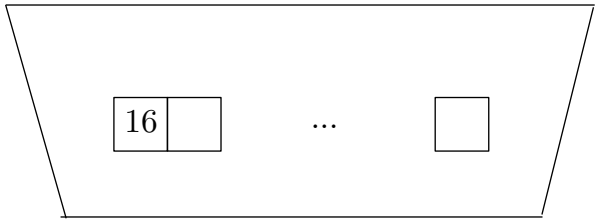
Example

Input:



38	28	88	17	26	41	72	83	69	47	12	68	5	52	35	9													

After the induction step at $n = 16$:



38	28	9	17	26	41	72	83	69	47	12	68	5	52	35	88													

sort these 15 elements recursively

Analysis of Selection Sort

Let $f(n)$ be the worst-case running time of selection sort when the problem size is n. From the base case, we know:

$$f(n) \quad = \quad O(1)$$

From the inductive case, we have:

$$f(n) \quad \leq \quad O(n) + f(n-1)$$

where the $O(n)$ term captures the cost of Steps 1 and 2, and $f(n-1)$ is the cost of Step 3.

Analysis of Selection Sort

So it remains to solve the recurrence (c_1, c_2 are constants whose values we do not care):

$$\begin{aligned} f(1) &= c_1 \\ f(n) &\leq c_2 n + f(n-1) \end{aligned}$$

Using the expansion method, we get:

$$\begin{aligned} f(n) &\leq c_2 n + f(n-1) \\ &\leq c_2 n + c_2(n-1) + f(n-2) \\ &\leq c_2 n + c_2(n-1) + c_2(n-2) + f(n-3) \\ &\leq c_2 n + c_2(n-1) + \ldots + c_2 \cdot 2 + f(1) \\ &\leq c_2 n(n+1)/2 + c_1 \\ &= O(n^2). \end{aligned}$$

We now conclude that the selection sort algorithm solves the sorting problem in $O(n^2)$ worst-case time.

In this lecture, we have seen two applications of recursion. This technique's importance in computer science can never be overstated. It will be utilized repeatedly in this course. So will it in the rest of our lives.