Recursion: The Beginning

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

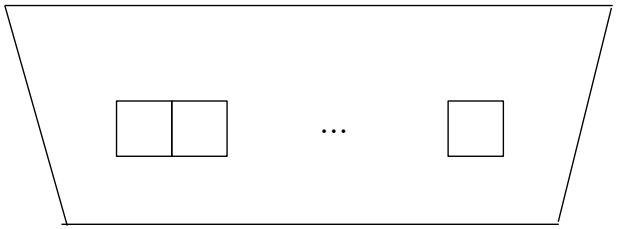
This lecture will introduce a useful technique called recursion. If used judiciously, this technique often leads to elegant algorithms that are easy to understand and analyze.

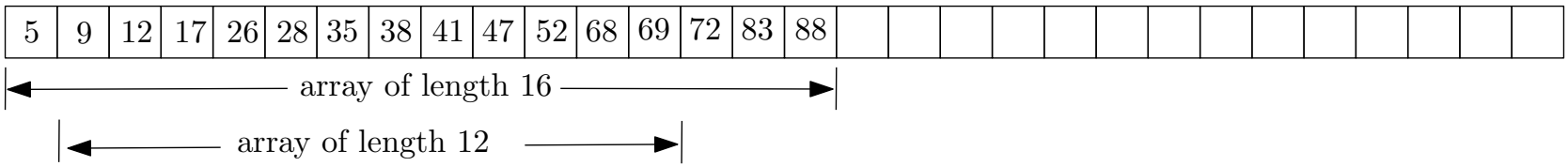
To make our point, we will first "re-discover" the binary search algorithm by way of recursion. Then, we will use the technique to obtain an algorithm for solving another problem called sorting. This will pave the way for more advanced applications of recursion in later lectures.

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 can be summarized as:

1. **[Reduce]**
   Show that if we can solve the same problem but with a size smaller than n, then we can solve the original problem (of size n).
2. **[Base]**
   When the problem size has become sufficiently small, solve it trivially.

Binary Search (Re-discovered)

Reduce (Problem Size $n > 1$)

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.

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).

Our algorithm description is now complete!

Analysis of Binary Search

Next we will see how recursion allows us to analyze the running time in a neat way.

Define $f(n)$ to be the worst-case running time of binary search. From the base case, we know:

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

From the inductive case, we know:

$$f(n) \quad \leq \quad O(1) + f(n/2)$$

where the $O(1)$ term is the cost of Step 1 (of Slide 6), and the $f(n/2)$ term is the cost of Step 2.1 or 2.2, noticing that each of the two steps runs binary search on a problem of size at most $n/2$.

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:

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

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 classical problem in computer science, and is worth several lectures' discussion.

The Sorting Problem

Problem Input:

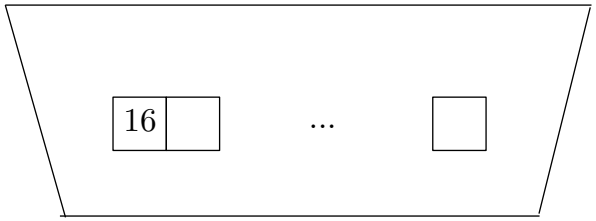
A set S of n integers is given in an array of length n. The value of n is inside the CPU (i.e., in a register).

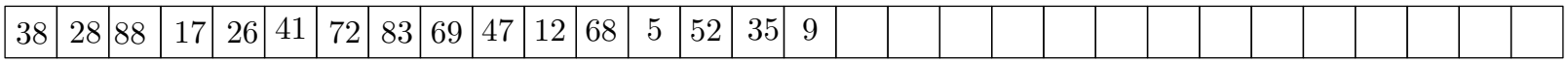
Goal:

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

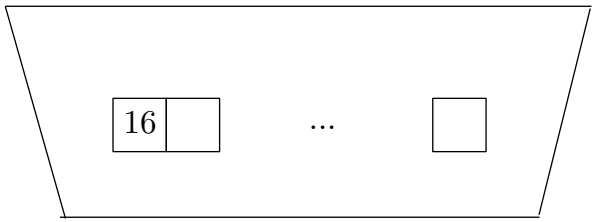
Example

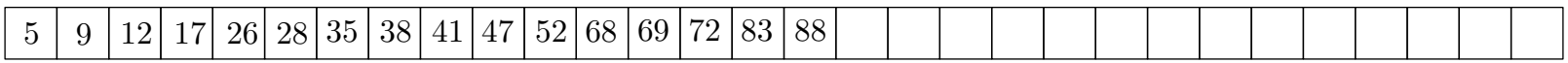
Input:





Output:





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

Selection Sort

Reduce(Problem Size $n > 1$)

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.

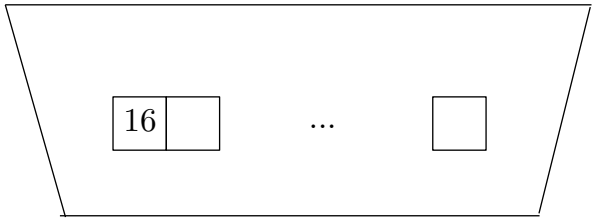
Selection Sort

Base ($n = 1$)

Trivial. Nothing to sort. Return directly.

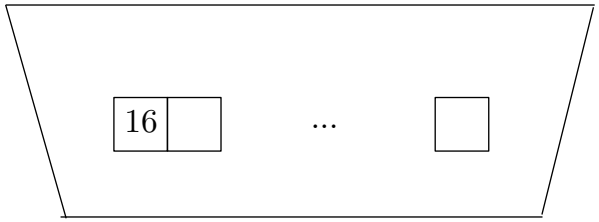
Example

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:

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

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