

Recursion (the Beginning)

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

This lecture will introduce a technique called **recursion** for designing algorithms. Its principle is:

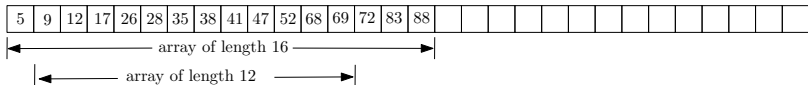
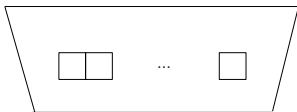
When dealing with a subproblem (same problem but with a smaller input), consider it solved, and use the subproblem's output to continue the algorithm design.

We will apply the technique to settle several problems in this course. Today, we will see two examples. In the first, we will re-discover binary search; in the second, we will design our first sorting algorithm.

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** .

Binary Search (Re-discovered)

1. Compare v to the middle element e of the array. If $v = e$, return “yes” and done.
2. Otherwise:
 - 2.1 If $v < e$, we have a **subproblem**: check if v is in the portion of the array before e ;
 - 2.2 If $v > e$, we have a **subproblem**: check if v is in the portion of the array after e .

Considering the subproblem solved, we finish the algorithm.

Think: why does it work?

Analysis of Binary Search

Recursion allows us to analyze the running time in an elegant manner.

Define $f(n)$ to be the maximum running time of binary search on n elements. For $n = 1$, clearly:

$$f(1) = O(1)$$

For $n > 1$:

$$f(n) \leq O(1) + f(\lfloor n/2 \rfloor).$$

Analysis of Binary Search

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

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

Suppose, for now, that n is a power of 2. An easy way of doing so is the **expansion method**, which simply expands $f(n)$ all the way down:

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

Analysis of Binary Search

We can deal with general n (not necessarily a power of 2) using a **rounding** approach. 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:

Produce an array that stores the elements of S in **ascending order**.

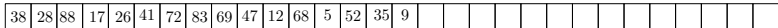
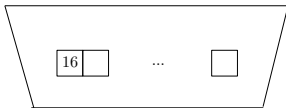
Selection Sort

1. Find the largest integer e_{max} in S .
2. Swap e_{max} with the last (i.e., n -th) element of the array (after which e_{max} is at the end of the array).
3. We now have a **subproblem**: sort the first $n - 1$ elements.

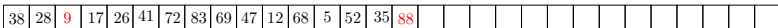
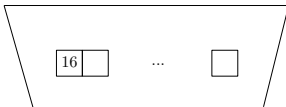
Let us consider that the subproblem has been solved. Now, the entire array is in ascending order. We thus finish the algorithm.

Example

Input:



After Step 2:



← sort these 15 elements recursively →

Analysis of Selection Sort

Let $f(n)$ be the maximum running time of selection sort when the problem size is n . We know:

$$f(1) = O(1)$$

For $n \geq 2$, we have:

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

where the term $O(n)$ 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):

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

Using the expansion method, we get:

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

We now conclude that selection sort runs in $O(n^2)$ worst-case time.