Algorithms: Divide-and-Conquer (Merge-Sort)

Alessandro Chiesa, Ola Svensson

EPFL

School of Computer and Communication Sciences

Lecture 3, 25.02.2025

Recall Last Lecture: Loop Invariant

CalculateSum(n):

1. $ans = 0$
2. **for** $i = 1, 2, \ldots, n$
3. $\quad ans = ans + i$
4. **return** ans

Often used for proof of correctness in presence of loops

Loop invariant = "a statement that is satisfied during the loop"

Ex: At the start of each iteration $ans = (i-1) * i/2$

Need to verify (similar to induction)

Initialization: True at the beginning of the 1st iteration of the loop

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant — usually along with the reason that the loop terminated — gives us a useful property that helps show that the algorithm is correct.

The difficulty is often to come up with the right loop invariant

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

```
Linear-Search(A,v)
    for i ← 1 to length(A)
        if A[i] = v then
            return i
    return NIL
```

Loop invariant:

At the start of each iteration of the “outer” **for** loop – the loop indexed by j – the subarray $A[1\ldots,j-1]$ consists of the elements originally in $A[1,\ldots,j-1]$ but in sorted order.

Loop invariant:

At the start of each iteration of the **for** loop we have $A[j] \neq v$ for all $j < i$.

Recall Last Lecture: Time Analysis

Random-access machine (RAM) model

- Instructions are executed one after another
- Simplification basic instructions take constant ($O(1)$) time
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return

Running time: on a particular input, it is the number of primitive operations (steps) executed

We usually concentrate on finding the **worst-case running time:** the longest running time for *any* input of size n

Order of growth: Focus on the important features

- Drop lower-order terms
- Ignore the constant coefficient in the leading term

Recall Last Lecture: Analysis of insertion sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,.\,.\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

number of times line executed based on the value of j

Worst case: The array is in reverse sorted

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \frac{n(n+1)-2}{2}$$

$$+ (c_6 + c_7)\frac{n \cdot (n-1)}{2} + c_8(n-1) = \Theta(n^2)$$

DIVIDE-AND-CONQUER

Merge Sort

Divide-and-Conquer

Powerful algorithmic approach:

recursively divide problem into smaller subproblems

Divide the problem into a number of subproblems that are smaller instances of the same problem

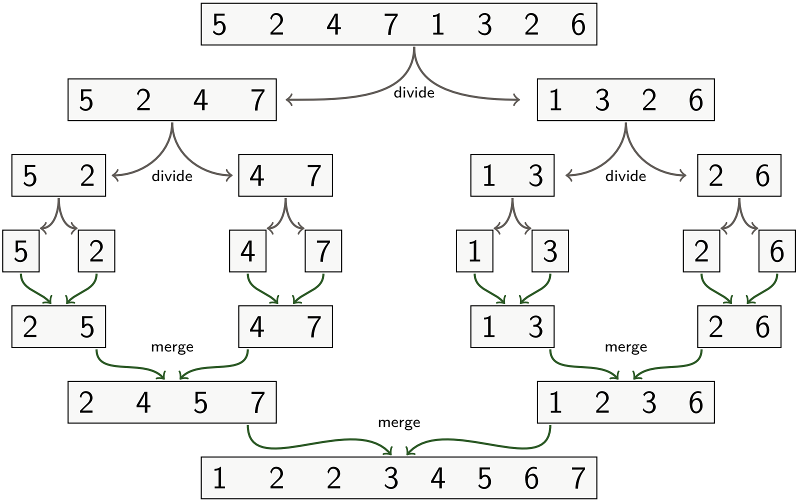
Conquer the subproblems by solving them recursively.

Base case: If the subproblems are small enough, just solve them by brute force

Combine the subproblem solutions to give a solution to the original problem

Merge Sort = D & C applied to sorting

Example $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$



Merge sort

To sort $A[p \ldots r]$:

Divide by splitting into two subarrays $A[p \ldots q]$ and $A[q+1, \ldots, r]$, where q is the halfway point of $A[p \ldots r]$

Conquer by recursively sorting the two subarrays $A[p \ldots q]$ and $A[q+1, \ldots r]$

Combine by merging the two sorted subarrays $A[p \ldots q]$ and $A[q+1, \ldots, r]$ to produce a singe sorted subarray $A[p \ldots r]$

```
MERGE-SORT(A, p, r)
  if p < r                        // check for base case
      q = ⌊(p + r)/2⌋             // divide
      MERGE-SORT(A, p, q)         // conquer
      MERGE-SORT(A, q + 1, r)     // conquer
      MERGE(A, p, q, r)           // combine
```

Merging

What remains is the MERGE procedure to solve the “merge” problem:

Definition

INPUT: Array A and indices $p \leq q < r$ such that subarrays $A[p \dots q], A[q+1 \dots r]$ are sorted.

OUTPUT: The two subarrays are merged into a single sorted subarray in $A[p \dots r]$.

Example:

2	4	5	7	1	2	3	6

$\downarrow$

1	2	2	3	4	5	6	7

Correctness of Merge-Sort

Assuming MERGE is correct

```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

Theorem

Assuming that the implementation of the MERGE *procedure is correct,* MERGE-SORT*(A,p,r) correctly sorts the numbers in* $A[p \ldots r]$

Proof by induction on $n = r - p$

Base case $n = 0$: In this case $r = p$ so $A[p \ldots r]$ is trivially sorted.

Inductive case: Assume statement true $\forall n \in \{0, 1, \ldots, k-1\}$ and prove the statement for $n = k$.

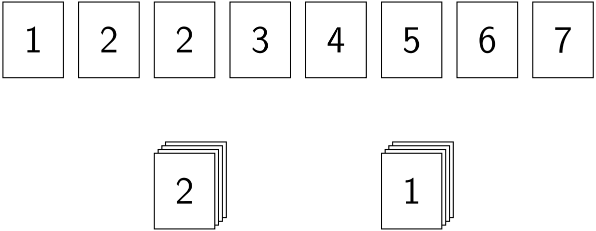
- By induction hypothesis MERGE-SORT(A, p, q) and MERGE-SORT(A, q+1, r) successfully sort the two subarrays.
- Therefore a correct merge procedure will successfully sort $A[p \ldots q]$ as required.

	p			q				r	
$\cdots$	1	2	2	3	4	5	6	7	$\cdots$

Idea behind linear-time merging

Think of two pile of cards that are placed face up

- Basic step: pick the smaller of the two cards and place it in the output pile



Idea behind linear-time merging

Think of two pile of cards that are placed face up

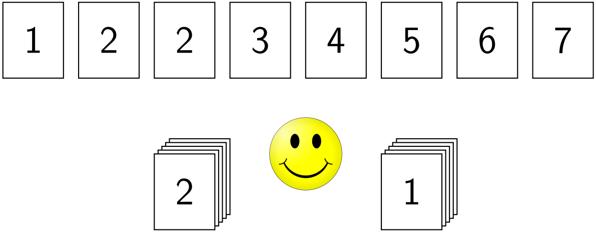
- Basic step: pick the smaller of the two cards and place it in the output pile
- There are $\leq n$ basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input pile
- Therefore the procedure should take $\theta(n)$ time

1	2	2	3	4	5	6	7

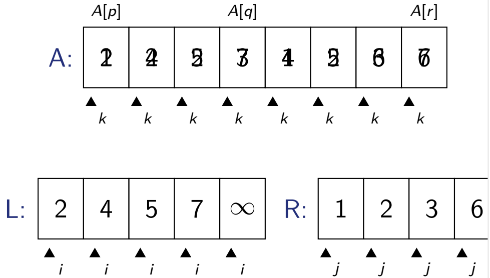
Implementation Simplification

Instead of checking whether a pile is empty:

- Put in the bottom of each input pile a special **sentinel** card of value ∞
- Stop once we have performed $n = r - p + 1$ basic steps (picked n cards)



Merging Algorithm



MERGE(A, p, q, r)

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,.\,.\, n_1 + 1]$ and $R[1 \,.\,.\, n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
  $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
  $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$
    $i = i + 1$
  else $A[k] = R[j]$
    $j = j + 1$

MERGE(A, p, q, r)

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,.\,.\, n_1 + 1]$ and $R[1 \,.\,.\, n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
  $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
  $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$

Merging Algorithm

- Runtime analysis?

MERGE(A, p, q, r)

```
n_1 = q - p + 1
n_2 = r - q
let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
for i = 1 to n_1
    L[i] = A[p + i - 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Analyzing divide-and-conquer algorithms

Use a recurrence equation to describe the running time:

- Let $T(n) =$ "running time on a problem of size n"
- If n is small enough say $n \leq c$ for some constant c then $T(n) = \Theta(1)$ (by brute force)
- Otherwise, suppose we divide into a sub problems each of size n/b.
- Let $D(n)$ be the time to divide and let $C(n)$ the time to combine solutions.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

Analysis of Merge Sort

```
MERGE-SORT(A, p, r)
  if p < r                          // check for base case
      q = ⌊(p + r)/2⌋               // divide
      MERGE-SORT(A, p, q)           // conquer
      MERGE-SORT(A, q + 1, r)       // conquer
      MERGE(A, p, q, r)             // combine
```

Divide: takes constant time, i.e., $D(n) = \Theta(1)$

Conquer: recursively solve two subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: Merge on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

Comparing the Two Sorting Algorithms

	worst-case running time	in-place
Insertion Sort	$\Theta(n^2)$	YES
Merge Sort	$\Theta(n \log n)$	NO

- A sorting algorithm is in-place if the numbers are rearranged within the array (while using at most a constant amount of additional space)
- Insertion sort is incremental: having sorted the subarray $A[1 \ldots j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1 \ldots j]$.
- Merge sort is divide-and-conquer: break the problem into smaller subproblems and then combine the solutions to the subproblems

SOLVING RECURRENCES

INDUCTION

INDUCTION

INDUCTION

INDUCTION

INDUCTION

INDUCTION

INDUCTION

INDUCTION

INDUCTION

INDUCTION

Analysing Recurrences

As an example, we shall consider the following recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + c \cdot n & \text{otherwise.} \end{cases}$$

Note that this recurrence upper bounds and lower bounds the recurrence for MERGE-SORT by selecting c sufficiently large and small, respectively.

We shall solve recurrences by using three techniques:

- The substitution method
- Recursion trees
- Master method

The substitution method

- Guess the form of the solution
- Use mathematical induction to find the constants and show that the solution works.

$$T(n) = 2T(n/2) + c \cdot n$$

A qualified guess is that $T(n) = \Theta(n \log n)$

The substitution method: proof of guess

Upper bound

There exists a constant $a > 0$ such that $T(n) \leq a \cdot n \log n$ for all $n \geq 2$

Proof by induction on n

Base cases: For any constant $n \in \{2, 3, 4\}$, $T(n)$ has a constant value, selecting a larger than this value will satisfy the base cases when $n \in \{2, 3, 4\}$.

Inductive step: Assume statement true $\forall n \in \{2, 3, \ldots, k-1\}$ and prove the statement for $n = k$.

$$T(n) = 2T(n/2) + cn$$

We can thus select a to be a positive constant so that both the base cases and the inductive step holds. Hence, $T(n) = O(n \log n)$

The substitution method: proof of guess

Lower bound

There exists a constant $b > 0$ such that $T(n) \geq b \cdot n \log n$ for all $n \geq 0$

Proof by induction on n

Base case: For $n = 1$, $T(n) = c$ and $b \cdot n \log n = 0$ so the base case is satisfied for any b.

Inductive step: Assume statement true $\forall n \in \{0, 1, \ldots, k-1\}$ and prove the statement for $n = k$.

$$T(n) = 2T(n/2) + cn$$

We can thus select b to be a positive constant so that both the base cases and the inductive step holds. Hence, $T(n) = \Omega(n \log n)$

Common mistake using the substitution method

Be careful when using asymptotic notation!

The false proof for the recurrence $T(n) = 4T(n/4) + n$, that $T(n) = O(n)$:

$$\begin{aligned} T(n) &\leq 4(c(n/4)) + n \\ &\leq cn + n = O(n) \qquad \text{wrong!} \end{aligned}$$

Because we haven't proven the *exact form* of our inductive hypothesis (which is that $T(n) \leq cn$), this proof is false

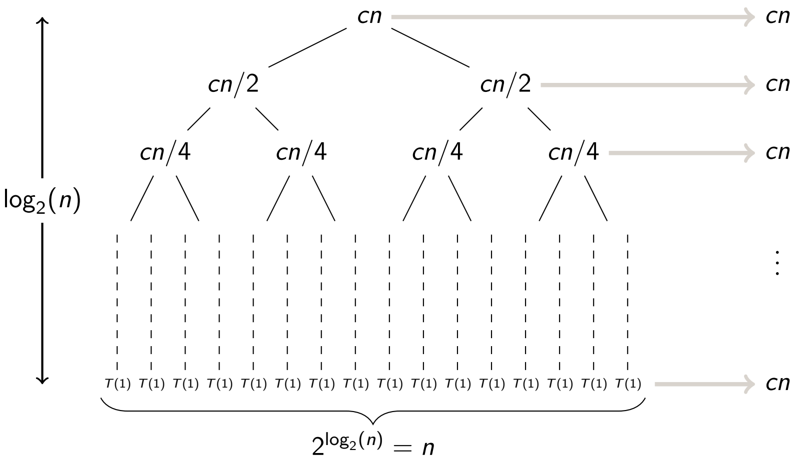
Recursion trees

Another way to generate a guess. Then verify by substitution method.

- Each node corresponds to the cost of a subproblem
- We sum the costs within each level of the tree to obtain a set of per-level costs,
- then we sum all the per-level costs to determine the total cost of all levels of the recursion.

Recursion trees

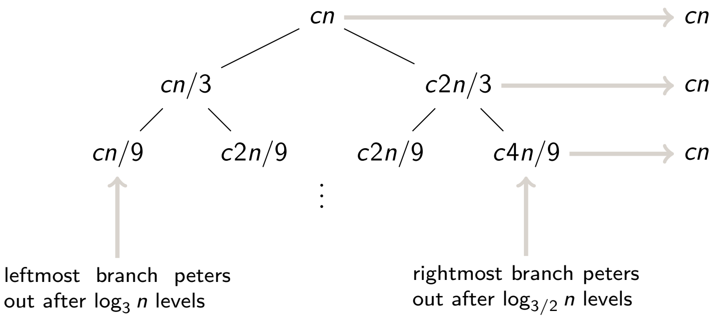
Our favorite example: $T(1) = c$ and $T(n) = 2T(n/2) + cn$



Qualified guess: $T(n) = cn \log_2 n = \Theta(n \log n)$

Recursion trees

Another interesting example: $T(n) = T(n/3) + T(2n/3) + cn$



- There are $\log_3 n$ full levels and after $\log_{3/2} n$ levels the problem size is down to 1.
- Each level contributes $\approx cn$

Qualified guess: exist positive constants a, b so that

$$a \cdot n \log_3(n) \leq T(n) \leq b \cdot n \log_{3/2} n \Rightarrow T(n) = \Theta(n \log n)$$

Master method

Used to black-box solve recurrences of the form $T(n) = aT(n/b) + f(n)$

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n).$$

Then, $T(n)$ has the following asymptotic bounds

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$*
- *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$*

Our favorite example: $T(1) = c$ and $T(n) = 2T(n/2) + cn$

- $f(n) = O(n)$ and $a = b = 2$ so $\log_b(a) = 1$ and $f(n) = \Theta(n^{\log_b(a)})$.
- By Master theorem, we have $T(n) = \Theta(n \log n)$:)

Summary

- Divide-and-conquer simple but powerful algorithmic paradigm
- Solving the recurrence for merge sort shows that it runs in time $\Theta(n \log n)$, i.e., much faster than Insertion sort for large instances
- For small instances insertion sort can still be faster
- Solving recurrences fun but delicate