

Chapter 2 - Getting started [Cormen]

- Question: How can we analyze algorithms?

↳ Behaviour?
↳ Time?
↳ Etc...

- Let's look at an example...

- Problem: Sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Algorithm: Insertion sort

- Idea: Sequentially compare an element against all other elements

↓
1st step: 5 2 4 6 1 3 for $j=2$ to length[A]
2nd step: 2 5 4 6 1 3 do $key = A[j]$
3rd step: 2 4 5 6 1 3 $i = j-1$
4th step: 2 4 5 6 1 3 while $i > 0$ and $A[i] > key$
5th step: 1 2 4 5 6 3 $A[i+1] = A[i]$
6th step: 1 2 3 4 5 6 $i = i-1$
 $A[i+1] = key$

Question: How much time does the algorithm require to solve a task?

- Different computers have different running times:

- CPU
- RAM
- Cache
- Bus
- Load
- Etc

- The same computer has different running times
 - Load
 - OS
 - cache
 - Bus
 - Etc...

• We need to represent ~~this~~ the overall "time" in a manner that is independent of these variables...

• Let's look again at insertion sort:

Cost to execute the instruction on a certain machine

Insertion Sort (A)

Cost Times

for $j = 2 : \text{length}[A]$

C_1

n

do $\text{key} = A[j]$

C_2

$n-1$

$i = j - 1$

C_3

$n-1$

while $i > 0$ and $A[i] > \text{key}$

C_4

$\sum_{j=2}^n t_j$

do $A[i+1] = A[i]$

C_5

$\sum_{j=2}^n (t_j - 1)$

$i = i - 1$

C_6

$\sum_{j=2}^n (t_j - 1)$

$A[i+1] = \text{key}$

C_7

$n-1$

* t_j = the number of times the while loop is executed for that value of j

• Question: What is the total running time to sort n elements?

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=1}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_8 (n-1)$$

- The running time depends on:

- the size of the input: n

- which input is given:

If the array is already sorted there will be intrinsically less operations to execute...

If the array is completely reversed there will be a lot of operations to execute...

- The best running time happens when all elements are already sorted:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_5 (n-1) \\ = (c_1 + c_2 + c_3 + c_4 + c_5) n + (c_2 + c_3 + c_4 + c_5)(n-1)$$

- This running time can be expressed as $an + b$ and is thus a linear function of n

- If the array is in reverse sorted order the worst case results

- We must compare each element $A[j]$ with each element in the entire ~~array~~ subarray $A[1 \dots j-1]$ and so $t_j = j$ for $j = 2, 3, \dots, n$

$$\sum_{j=2}^n t_j = \sum_{j=2}^n j = \frac{1}{2} n (n+1) - 1$$

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n (j-1) = \frac{1}{2} n (n-1)$$

Notes:

Arithmetic series:

$$\sum_{k=1}^n k = \frac{1}{2} n (n+1)$$

See the included notes of the Arithmetic Series

Arithmetic Series

- Sequence of numbers such that the difference between the consecutive terms is constant

$$\left. \begin{aligned} - a_n &= a_1 + (n-1)d \\ - a_n &= a_m + (n-m)d \end{aligned} \right\} \begin{aligned} a_1 &= \text{first term} \\ a_n &= \text{last term} \\ a_m &= \text{the } m^{\text{th}} \text{ term} \\ d &= \text{constant difference} \end{aligned}$$

If we use the normal sum then:

$$- S_n = a_1 + (a_1 + d) + (a_1 + d) + \dots + (a_1 + (n-2)d) + (a_1 + (n-1)d)$$

If instead we start summing from the end

$$- S_n = (a_n - (n-1)d) + (a_n - (n-2)d) + \dots + (a_n - 2d) + (a_n - d)$$

Adding both sides of the equation we get

$$2S_n = n(a_1 + a_n) \quad \left[\begin{array}{l} \text{All terms involving } d \\ \text{are cancelled} \end{array} \right]$$

$$\Rightarrow S_n = \frac{n}{2} (a_1 + a_n)$$

$$\Rightarrow S_n = \frac{n}{2} [a_1 + (a_1 + (n-1)d)]$$

$$\Rightarrow S_n = \frac{n}{2} [2a_1 + (n-1)d]$$

But this always assumes that the initial index of the sum is 1.

Question: What happens when $k > 1$?

Let's look at an example:

$$\sum_{j=1}^n 2j \quad \begin{array}{ccccc} j=1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \end{array}$$

$$\sum_{j=1}^n 2j \quad \begin{array}{ccccc} j=5 & 4 & 3 & 2 & 1 \\ 10 & 8 & 6 & 4 & 2 \end{array}$$

$$\begin{array}{ccccc} 2 & 4 & 6 & 8 & 10 \\ + 10 & 8 & 6 & 4 & 2 \\ \hline 12 & 12 & 12 & 12 & 12 \end{array}$$

Question:

Why $1/2$? Because we are using $\sum_{j=1}^n 2j$ and $\sum_{j=1}^n 2j$ thus we will have double the initial value

$$\begin{aligned} & \frac{1}{2} (a_1 + a_n) \cdot n = \\ & = \frac{1}{2} (a_1 + a_n) (n - (1 + 1)) \end{aligned}$$

$$\boxed{S = \frac{1}{2} (a_1 + a_n) (n - \text{initial Index} + 1)}$$

Let's see if it works:

Let's calculate manually: $\sum_{j=1}^n 2j \stackrel{n=5}{=} 2 + 4 + 6 + 8 + 10 = 30$

Now using the derived formula: $\sum_{j=1}^n 2j \stackrel{n=5}{=} \frac{1}{2} (2 + 2 \cdot 5) (5 - 1 + 1) = \frac{1 \cdot 12 \cdot 5}{2} = 30$

It works $\checkmark$

$$\begin{aligned}
 T(n) &= c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \left[\frac{n(n+1)}{2} \cdot n \right] \\
 &\quad + c_5 \left[\frac{n(n-1)}{2} \right] + c_6 \left[\frac{n(n-1)}{2} \right] + c_7 (n-1) \\
 &= \left[\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right] n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) n \\
 &\quad - (c_2 + c_3 + c_4 + c_7)
 \end{aligned}$$

- This worst-case running time can be expressed as $an^2 + bn + c$ and is thus a quadratic function of n .

Question: With what metric do you think we should be dealing with?

↳ Best-case?

↳ Worst-case?

- The worst-case running time of an algorithm is an upper-bound of the running time for any input.
 - the algorithm will never take longer...
 - Allows us to make an educated guess...

2.3 Designing Algorithms

- Insertion sort used what is called an incremental technique
- There is also the divide and conquer approach
 - Recursive in nature...

- The divide-and-conquer paradigm:

- Divide: the problem into subproblems...

- Conquer: the subproblems by solving them recursively

- Combine: the solutions to the subproblems into the solution for the original problem

- The merge sort algorithm uses a divide-and-conquer approach

Divide: Divide the n -element sequence to be sorted into two subsequences of $n/2$ elements each

Conquer: Sort the two subsequences recursively using merge sort

Combine: Merge the two sorted subsequences to produce the sorted answer

The recursion bottoms out when the sequence to be sorted has length 1

- The key operation of the merge sort algorithm is the merging of two sorted ~~seq~~ sequences:

- Merge (A, p, q, r)

- A = array

- p, q, r are index numbering elements of the array such that $p \leq q < r$

- Assumes that $A[p \dots q]$ and $A[q+1 \dots r]$ are sorted

- It merges them to form a single sorted subarray and replaces the current subarray $A[p \dots r]$

Merge (A, p, q, r)

Line 1: $n_1 = q - p + 1$ /* How many elements to the left? */

" 2: $n_2 = r - q$ /* How many elements to the right? */

" 3: $L = [1 \dots n_1 + 1]$

" 4: $R = [1 \dots n_2 + 1]$

Line 5: for $i = 1$ to n_1 : $L[i] = A[p + i - 1]$

" 6: for $j = 1$ to n_2 : $R[j] = A[q + j]$

Line 7: $L[n_1 + 1] = \infty$ [sentinel value]

" 8: $R[n_2 + 1] = \infty$ [sentinel value]

...

$i = j = 1$ index for the left array : line 2

index for the right array : line 3

for $k = p$ to r : line 10

if $L[i] \leq R[j]$: line 11

$A[k] = L[i]$: line 12

$i = i + 1$: line 13

else : line 14

Line 15: $A[k] = R[j]$

Line 16: $j = j + 1$

Merge (A, 8, 12, 16)

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \dots & 2 & 4 & 5 & 7 & 1 & 2 & 3 & 6 & \dots \\ \hline \end{array}$$

$$L = \begin{array}{|c|c|c|c|c|} \hline 2 & 4 & 5 & 7 & \infty \\ \hline \end{array}$$

$$R = \begin{array}{|c|c|c|c|c|} \hline 1 & 2 & 3 & 6 & \infty \\ \hline \end{array}$$

1st step:

$$\begin{array}{l} A = 1 \quad 4 \quad 5 \quad 7 \quad 1 \quad 2 \quad 3 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

2nd step:

$$\begin{array}{l} A = 1 \quad 2 \quad 5 \quad 7 \quad 1 \quad 2 \quad 3 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

3rd step:

$$\begin{array}{l} A = 1 \quad 2 \quad 7 \quad 1 \quad 2 \quad 3 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

4th step:

$$\begin{array}{l} A = 1 \quad 2 \quad 3 \quad 1 \quad 2 \quad 3 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

5th step:

$$\begin{array}{l} A = 1 \quad 2 \quad 3 \quad 4 \quad 2 \quad 3 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

6th step:

$$\begin{array}{l} A = 1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 5 \quad 3 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

7th step:

$$\begin{array}{l} A = 1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 6 \\ L = 2 \quad 4 \quad 5 \quad 7 \quad \infty \\ R = 1 \quad 2 \quad 3 \quad 6 \quad \infty \end{array}$$

3th step: $A = 1 \ 2 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7$ 7/
 $L = 2 \ 4 \ 5 \ 7 \ 00$
 $R = 1 \ 2 \ 3 \ 6 \ 00$

- We can now define the algorithm:

MergeSort (A, p, r)

if ($p < r$) ** Always check boundary conditions in a recursive **

$q = \left\lfloor \frac{p+r}{2} \right\rfloor$ (floor)

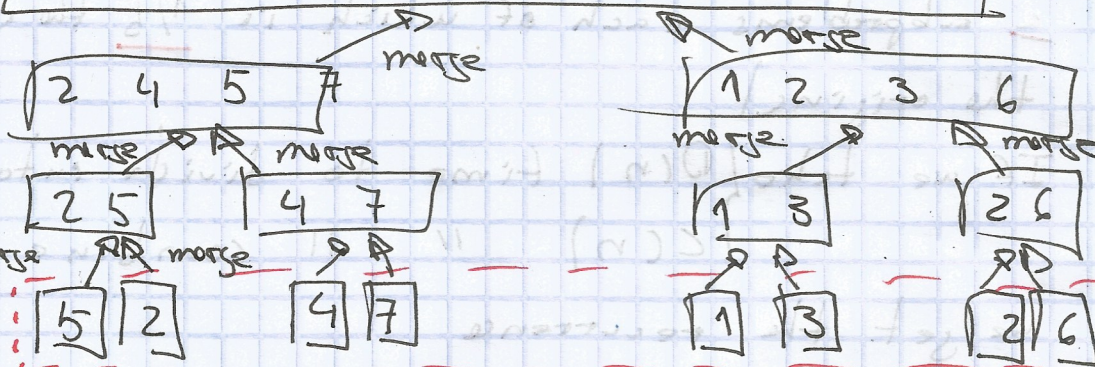
MergeSort (A, p, q)

MergeSort ($A, q+1, r$)

Merge (A, p, q, r)

- Example: For clarification also see the function call of page 7.1

1 2 2 3 4 5 6 7



Initial Array: [5] [2] [4] [7]

↳ After the recursive procedure "bottomout"...

Question: How much time does Merge Sort require?

- Merge procedure runs in $\Theta(n)$

Lines 1-4: constant time c_1

Lines 7-9: constant time c_2

Line 10: $n_1 + n_2 = n$ time

Lines 11-16: constant time c_3

$c_1 + c_2 + n + n c_3$

$c_1 + c_2 + 2n$

(linear time) $= \Theta(n)$

If we ignore the constant

- What about merge sort?

A =

5	2	4	7	4	3	2	6
---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8

7.1

Merge Sort (A, 1, 8)

$q=4$
Merge Sort (A, 1, 4)

$q=2$
Merge Sort (A, 1, 2)

$q=1$
Merge Sort (A, 1, 1) /* Returns */

Merge Sort (A, 2, 2) /* Returns */

Merge (A, 1, 1, 2) /* Merges

2	5
---	---

 */

Merge Sort (A, 3, 4)

$q=3$
Merge Sort (A, 3, 3) /* Returns */

Merge Sort (A, 4, 4) /* Returns */

Merge (A, 3, 3, 4) /* Merges

4	7
---	---

 */

Merge (A, 1, 2, 4) /*

2	4	5	7
---	---	---	---

 */

Merge Sort (A, 5, 8)

$q=6$
Merge Sort (A, 5, 6)

$q=5$
Merge Sort (A, 5, 5) /* Returns */

Merge Sort (A, 6, 6) /* Returns */

Merge (A, 5, 5, 6) /* Merges

1	3
---	---

 */

Merge Sort (A, 7, 8)

$q=7$

Merge Sort (A, 7, 7) /* Returns */

Merge Sort (A, 8, 8) /* Returns */

Merge (A, 7, 7, 8) /* Merges

2	6
---	---

 */

Merges (A, 5, 6, 8) /* Merges

1	2	3	6
---	---	---	---

 */

Merge (A, 1, 4, 8) /* Merges

1	2	2	3	4	5	6	7
---	---	---	---	---	---	---	---

 */

2.3.2 Analyzing divide-and-conquer algorithms

- Recursive algorithms can often be analyzed by a recurrence equation
- Recurrence equation describe the overall running time on a problem of size n in terms of the running time on smaller inputs
- Let $T(n)$ be the running time on a problem of size n
 - If the problem is small enough, say $n \leq c$, for some constant c , the straightforward solution takes ~~the~~ constant time, which we write as $\Theta(1)$
 - Suppose that our division of the problem yields b subproblems each of which is $1/b$ the size of the original
 - If we take $\begin{cases} D(n) & \text{time to divide into subproblems} \\ C(n) & \text{time to combine} \end{cases}$

we get the recurrence

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

- In the case of Merge Sort

Divide: Compute the middle of the array $\Rightarrow \Theta(1)$ time

Conquer: Recursively solve two subproblems each of size $\frac{n}{2}$
 $\Rightarrow 2T(\frac{n}{2})$

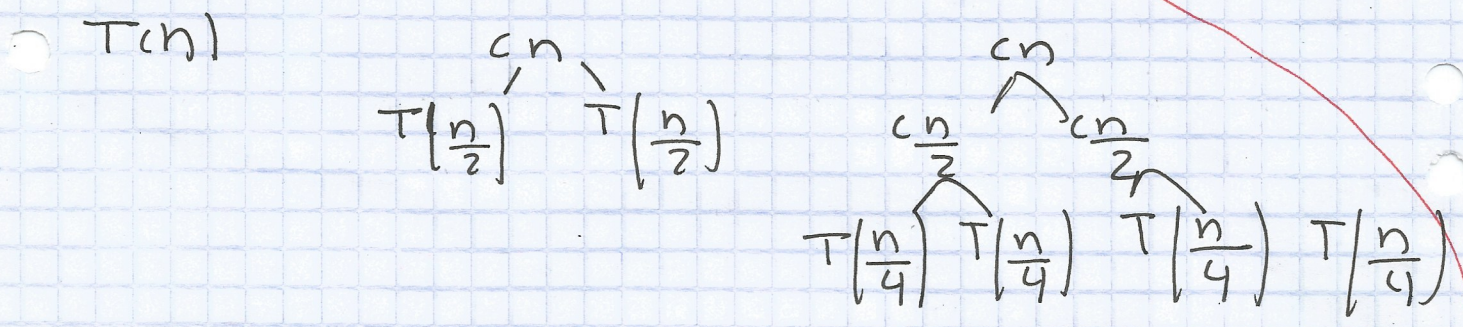
Combine: Merge the arrays $\Rightarrow \Theta(n)$ time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + \underbrace{\Theta(1) + \Theta(n)}_{\Theta(n)}, n > 1 \end{cases} = \begin{cases} \Theta(1), & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

Let us rewrite the recurrence as

$$T(n) = \begin{cases} c, & n=1 \\ 2T\left(\frac{n}{2}\right) + cn, & n > 1 \end{cases}$$

time required to solve problems of size 1 as well as the time per array element of the divide and combine algorithm



Eventually our tree will bottom-out in the recursion

- How many levels?
- Number of levels: $\log_2 n$ *Why?*

Cost of each level:

1st level: cn

2nd level: $\underbrace{\left(\frac{cn}{2} + \frac{cn}{2}\right)}_{cn}$

3rd level: $\underbrace{\left(\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4}\right)}_{cn}$

Total cost: Cost Each Level $\times$ Number of level

$$cn \times \log_2 n \Rightarrow \Theta(n \log n)$$

$\therefore$ Merge Sort will require time $\Theta(n \log n)$

0^{th} level: $n = \frac{n}{2^0}$
 1^{st} level: $n/2 = \frac{n}{2^1}$
 2^{nd} level: $n/4 = \frac{n}{2^2}$
 $\vdots$
 d^{th} level: $n/2^d$

What is the value of d such that $\frac{n}{2^d} = 1$ (i.e. when the recursion stops because input has size 1)

$$\frac{n}{2^d} = 1 \Leftrightarrow n = 2^d \Leftrightarrow d = \log_2 n$$

Exercises

2.3.3 Use mathematical induction to show that when n is an exact power of 2, the solution to the recurrence

$$T(n) = \begin{cases} \textcircled{2} & \text{if } n = 2 \text{ (base case)} \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

is $T(n) = \boxed{n \lg n}$

[resolution:]

Mathematical induction proofs always have two stages:

- 1st Prove the "base case"
- 2nd Prove the inductive step

Base case proof:

• $n = 2$ ~~we~~ $T(n) = n \lg n \stackrel{n=2}{\Rightarrow} 2 \lg 2 = 2 \cdot 1 = \textcircled{2}$

• Notice that the recursion case $T(n)$ for $n = 2$ is 2 so we have proved this result

Inductive step proof:

• Hypothesis: $T(n/2) = \boxed{\frac{n}{2} \lg \frac{n}{2}}$

Question: Why $n/2$?

→ With exact powers of 2, $n/2$ will always be an integer, i.e. an addressable input

• Hypothesis: $T(n/2) = (n/2) \lg(n/2)$

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

$= 2 \left[\frac{n}{2} \lg \frac{n}{2} \right] + n$

$= n \lg n/2 + n =$

$= n (\lg n - \lg 2) + n$

$= n (\lg n - 1) + n$

$= n \cdot \lg n - n + n = \boxed{n \lg n}$

Notes:

$\log u/v = \log u - \log v$

(end of proof)

Exercise 2.3.4

We can express insertion sort as a recursive algorithm as follows. In order to sort $A[1 \dots n]$ we recursively sort $A[1 \dots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \dots n-1]$. Write a recurrence for the running time of this recursive version of ~~insertion~~ sort.

[Resolution:]

Question: Eventually the recursion will have to sort a single element. How much time is required for sorting a single element?

We have no other elements to sort, just this one. Therefore we need a constant amount of time, i.e. $\Theta(1)$

- For the other cases we need to first sort $A[1 \dots n-1]$. Also notice that we only need to recurse on $A[1 \dots n-1]$, i.e. we are creating a single problem. Therefore:

$$D(n) = \Theta(1) \quad \left(\begin{array}{l} \text{we always divide the same way:} \\ - \text{start with } A[1 \dots n] \\ - A[1 \dots n-1] \\ - A[1 \dots n-2] \\ \vdots \\ - A[1 \dots 1] \end{array} \right) \quad \begin{array}{l} \text{Constant} \\ \text{Time} \\ \downarrow \\ \text{Just diminish} \\ \text{one "clot" every} \\ \text{step} \end{array}$$

~~$D(n)$~~

$a = 1$

Question: What about the time to combine $C(n)$?

- In the worst case, in order to insert $A[n]$ we need to go through the entire "subarray" that we are processing. At most we need to process n elements before ~~going to~~ inserting $A[n]$.
- This implies $O(n)$ time.

$$\therefore T(n) = \begin{cases} \Theta(1) & , \text{ if } n = 1 \\ T(n-1) + \Theta(n) & , \text{ if } n > 1 \end{cases}$$

Notes:

Recall that the general form of the recurrence equation is:

$$T(n) = \begin{cases} \Theta(n) & \text{if } n \leq c \\ a T\left(\frac{n}{b}\right) + \underbrace{D(n)}_{\text{Time to divide}} + \underbrace{C(n)}_{\text{Time to combine}}, & \text{otherwise} \end{cases}$$