# Algorithms

Alp Ozen

Fall 2020 - EPFL

# Contents

# 1 Week 1: Insertion sort, and simple analysis

**Insertion sort**

- Start with an empty hand of data
- Open data from left to right, each time placing it in the right order
- To find correct position, compare the data at hand with all other sorted data in the stack

**Code Snippet:**

```
for j = 2 to A.len
    key = A[j];
    i = j - 1;
    while i > 0 and A[i] > key
        A[i + 1] = A[i];
        i = i - 1;
    A[i + 1] = key;
```

**Example:**

Suppose we are given the array $8, 2, 5, 10, 1, 3$

Iteration 1: j=2 key = 2 i = 1

8,8

2,8

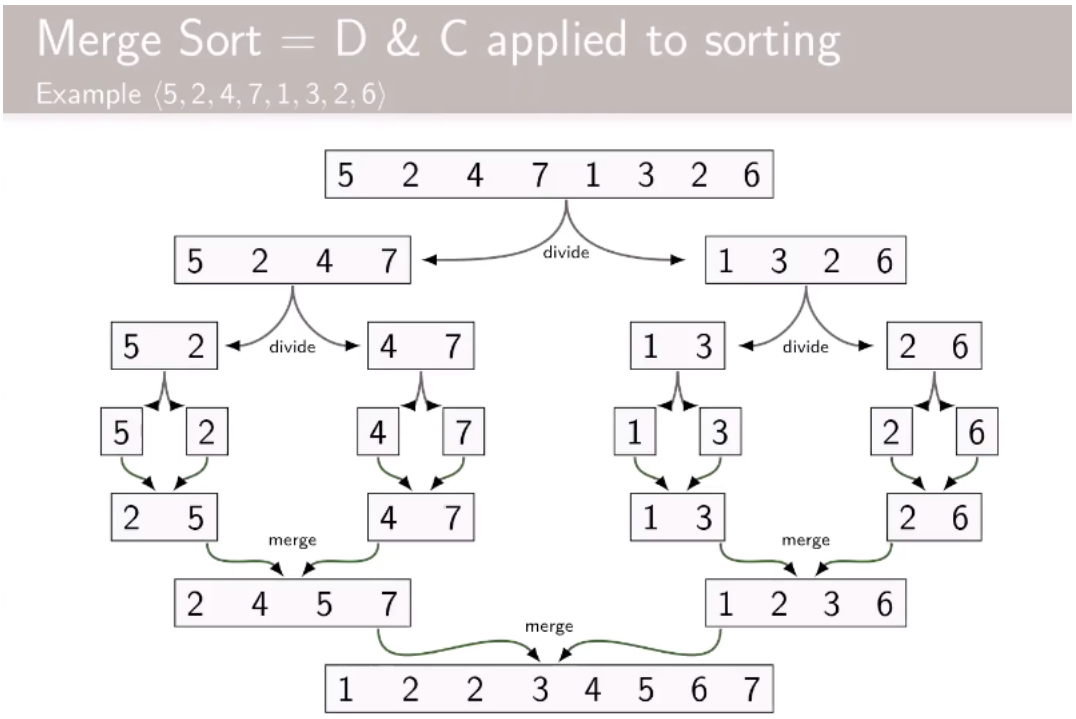
$\vdots$

Now we naturally ask how one may *prove* the correctness of an algorithm? To do this for insertion sort, we must show the **loop invariance** of our output array $A$. That is, the array $A$ must remain sorted during the initialization, maintenance and termination of the for loop. It is sorted at initialization because it is essentially an array of a single element. It is sorted at maintenance because each new element appended gets inserted into the correct place hence also why it is sorted at termination.

Another question to ask is the worst and best case running time of insertion sort. This of course depends on the input array. All lines up to the while loop are executed $n$ many times, $n$ being the length of the array. Then the while loop and the inside of it run some indeterminate $t$ many times. Yet this $t$ is simply how many times we have to shift an element to the right. Thus the best case runtime is $O(n)$ when given an already sorted array and the worst case is $O(n^2)$ when given an array in reverse order since at each step we do $j-1$ shifts resulting in $\sum_{j=2}^{n}(j-1)$.

The key takeaway from this week should be that an algorithm should both be minimal in its *runtime* while taking up minimal *space*.

## 2 Week 2: Divide Conquer: Merge sort




**Code Snippet:**

```
Merge_sort(A,p,r)
if(p<r)
    q = floor((p+r)/2)
    Merge_sort(A,p,q)
    Merge_sort(A,q+1,r)
    Merge(A,q,p,r)
```

We now need an algorithm for merging, as it turns out, we can provide a linear time algorithm for this feat.

**Code Snippet:**

```
Merge(A,p,q,r)
n_1 = q-p+1
n_2 = r-q
for i = 1 to n_1
L[i] = A[p+i-1]
for i = 1 to n_2
R[i] = A[q+i]
L[n_1 + 1] = inf
R[n_2 + 1] = inf
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
```

- Running time of merge is $\Theta(n \log n)$

Now let's prove the correctness of the entire algorithm inductively:

*Proof.* Let $n = r - p$.

**Base case:** $n = 0 \implies r = p$ hence array is trivially sorted.

**Inductive step:** Assume true $\forall n \in \{0, 1, \ldots, k-1\}$ Goal is to show that the procedure works for $n = k$

**Argument:** Well since we make the recursive calls $Merge-sort(A, p, q)$ and $Merge-sort(A, q+1, r)$ both ranges are in $\{0, 1, \ldots, k-1\}$ for which we know the procedure works, we are done. □

Let's now derive the runtime of merge-sort but also see a general way to prove the runtime of divide-conquer algorithms. Now let $D(n)$ denote the time to divide the problem, $C(n)$ the time to combine the problem and $T(n)$ the runtime on a problem of size $n$. Then we get the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(\frac{n}{b}) + C(n) + D(n) & \text{otherwise} \end{cases}$$

Now let's simplify. Observe that $D(n) = Theta(1)$ and that $C(n) = \Theta(n)$. Also since we subdivide the problem to size $\frac{n}{2}$ we have $2T(\frac{n}{2}) thus$ :

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

Now to solve this recurence there are three possible ways; **substitution method, recursion tree, master method**.

We first explore the *substitution method:*

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

**Example:**

$$T(n) = 2T(\frac{n}{2}) + cn$$

$$T(n) = 2(2T(\frac{n}{4}) + c\frac{n}{2}) = 4T(\frac{n}{4}) + 2cn$$

$$\vdots$$

This seems a lot like the general form is:

$$2^k T(\frac{n}{2^k}) + kcn$$

Now since $k = \log_2 n$, a qualified guess for $T(n)$ is that it is $\Theta(n \log n)$. To prove this supposition, we must show that the expression we obtained is both upper bounded, and lower bounded by $n \log n$

## 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 $n = k$.

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &\geq 2 \cdot \frac{bn}{2} \log(n/2) + c \cdot n = b \cdot n \log(n/2) + cn \\ &= b \cdot n \log n - bn + cn \\ &\geq b \cdot n \log n \qquad \text{(if we select } b \leq c\text{)} \end{aligned}$$

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

Lecture 3, 25.9.2020

## 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 $n \in \{2, 3, 4\}$.

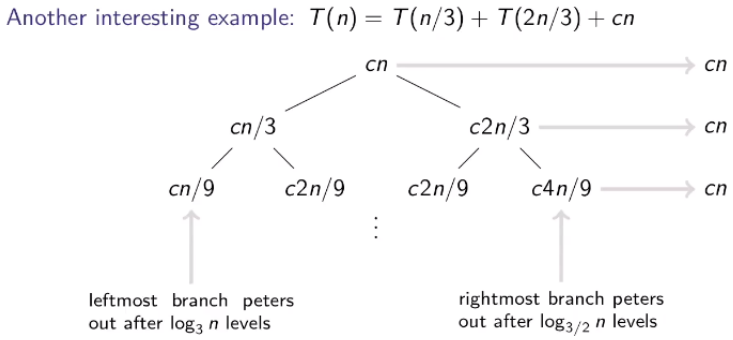
**Inductive step:** Assume statement true $\forall n \in \{2, 3, \ldots, k-1\}$ and prove the statement for $n = k$.

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &\leq 2 \cdot \frac{an}{2} \log(n/2) + c \cdot n = a \cdot n \log(n/2) + cn \\ &= a \cdot n \log n - an + cn \\ &\leq a \cdot n \log n \qquad \text{(if we select } a \geq c\text{)} \end{aligned}$$

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

Lecture 3, 25.9.2020

Now to make things more interesting, let's try to arrive at a runtime using a recursion tree approach.




As we can see, each level of the tree contributes some $cn$ steps to the runtime. Now we know that the shortest branch of the tree, the LHS will have $\log_3(n)$ branches and the RHS $\log_{\frac{3}{2}}(n)$ branches yielding a cost of $cn\log_3(n)$ in the worst case and $cn\log_{\frac{3}{2}}(n)$ in the best case.

We now present the master theorem without proof:
If we have some recurrence relation of form

$$T(n) = aT(\frac{n}{b}) + f(n)$$

, then the solution $\Theta(n)$ is either $f(n)$ or $n^{\log_b(a)}$ depending on which term dominates.