

CS 251 Homework 1

Dwijen Chawra (dchawra)

October 26, 2023

Problem 1.

Resources used: None

Collaborators: None

We need to prove that $\log(n!) = \Theta(n \log(n))$

This can be done by showing that the upper bound and lower bound are both $\Theta(n \log(n))$

We can refactor the original expression to look like so:

$$\log(n!) = \Theta(\log(n^n))$$

because

$$\begin{aligned}\log(n!) &= \log(n(n-1)(n-2)\dots(1)) \\ &= \log(n) + \log(n-1) + \log(n-2) + \dots + \log(1)\end{aligned}$$

We can see that the above expression is the same as the summation of $\log(n)$ from 1 to n .

Similarly,

Problem 2.

Resources used: None

Collaborators: None

a)

Base case: $F_1 = 3, F_2 = 4$ Inductive hypothesis: For all $n \geq 3$, $F_n \geq 2^{0.6n}$ and $F_{n-1} \geq 2^{0.6(n-1)}$

Inductive step:

$$F_{n+1} = 2^{0.6(n+1)}$$

$$F_n + F_{n-1} \geq F_{n+1}$$

We can infer that the minimum value of F_n and F_{n-1} is greater than or equal to the value of F_{n+1} .

$$2^{0.6n} + 2^{0.6(n-1)} \geq 2^{0.6(n+1)}$$

$$2^{0.6} + 1 \geq 2^{0.6 \cdot 2}$$

$$2.52... \geq 2.3...$$

We have proven the hypothesis with strong induction.

b)

Base case: $F_0 = 0, F_1 = 1$ Inductive hypothesis: For all $n \geq 0$, $F_n \leq 2^{0.7n}$ and $F_{n-1} \leq 2^{0.7(n-1)}$

Inductive step:

$$F_{n+1} = 2^{0.7(n+1)}$$

$$F_n + F_{n-1} \leq F_{n+1}$$

We can infer that the maximum value of F_N and F_{n-1} is less than or equal to the value of F_{n+1} .

$$\begin{aligned}2^{0.7n} + 2^{0.7(n-1)} &\leq 2^{0.7(n+1)} \\2^{0.7} + 1 &\leq 2^{0.7 \cdot 2} \\2.62\dots &\leq 2.63\dots\end{aligned}$$

We have proven the hypothesis with strong induction.

Problem 3.

Resources used: None

Collaborators: None

Base Case: 2, $T(2) = 2\log_2(2) = 2$

Inductive hypothesis: For all $k > 0$, $T(n) = n\log(n)$ if $n = 2^k$

Inductive step:

$$\begin{aligned}T(2^{k+1}) &= 2T(2^{k+1}/2) + 2^{k+1} \\&= 2T(2^{k+1}/2^1) + 2^{k+1} \\&= 2T(2^k) + 2^{k+1}\end{aligned}$$

Replace $T(2^k)$ with $2^k\log(2^k)$

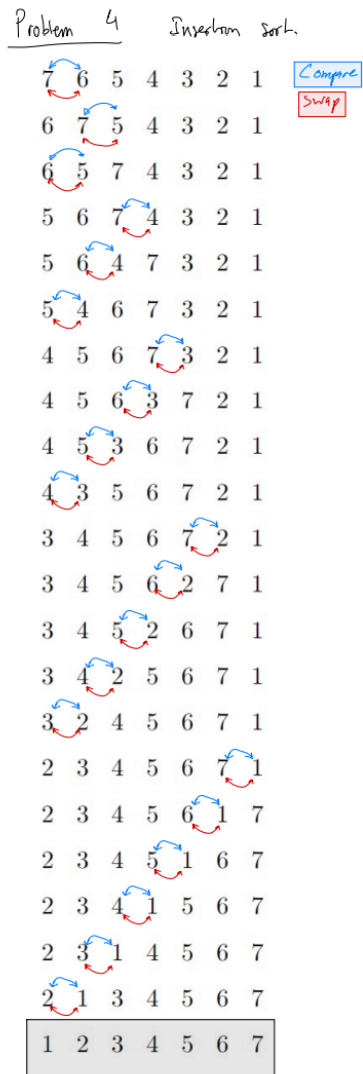
$$\begin{aligned}&= 2(2^k\log(2^k)) + 2^{k+1} \\&= 2^{k+1}\log(2^k) + 2^{k+1} \\&= 2^{k+1}(\log(2^k) + 1) \\&= 2^{k+1}\log(2^{k+1})\end{aligned}$$

We have proven the hypothesis with induction.

Problem 4.

Resources used: None — Collaborators: None

(a) 21 compares, 21 swaps.



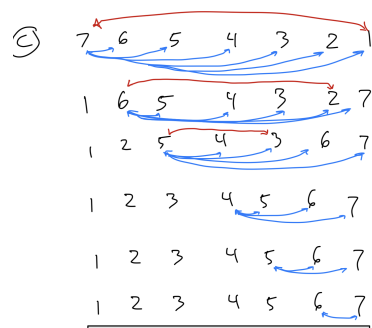
- (b) The number of swaps and compares for Insertion sort on an array $[n, n-1, \dots, 2, 1]$ would be as follows:

For first element, its compared to the previous, which results in 1 compare and one swap. The next one is compared to the previous two, which results in 2 compares and 2 swaps. This can be written with a summation.

$$\begin{aligned}\sum_{i=1}^{n-1} i &= \frac{n(n-1)}{2} \\ &= \frac{n^2 - n}{2} \\ &= \frac{n^2}{2} - \frac{n}{2}\end{aligned}$$

For swaps it is the exact same thing, since every compare results in a swap.

(c) 21 compares, 3 swaps.



- (d) Selection sort on an array $[n, n - 1, \dots, 2, 1]$ would have compares and swaps like this:

First run: Compares all elements from n to 1 to find the smallest one - n compares. Swaps the smallest element with the first element - 1 swap.

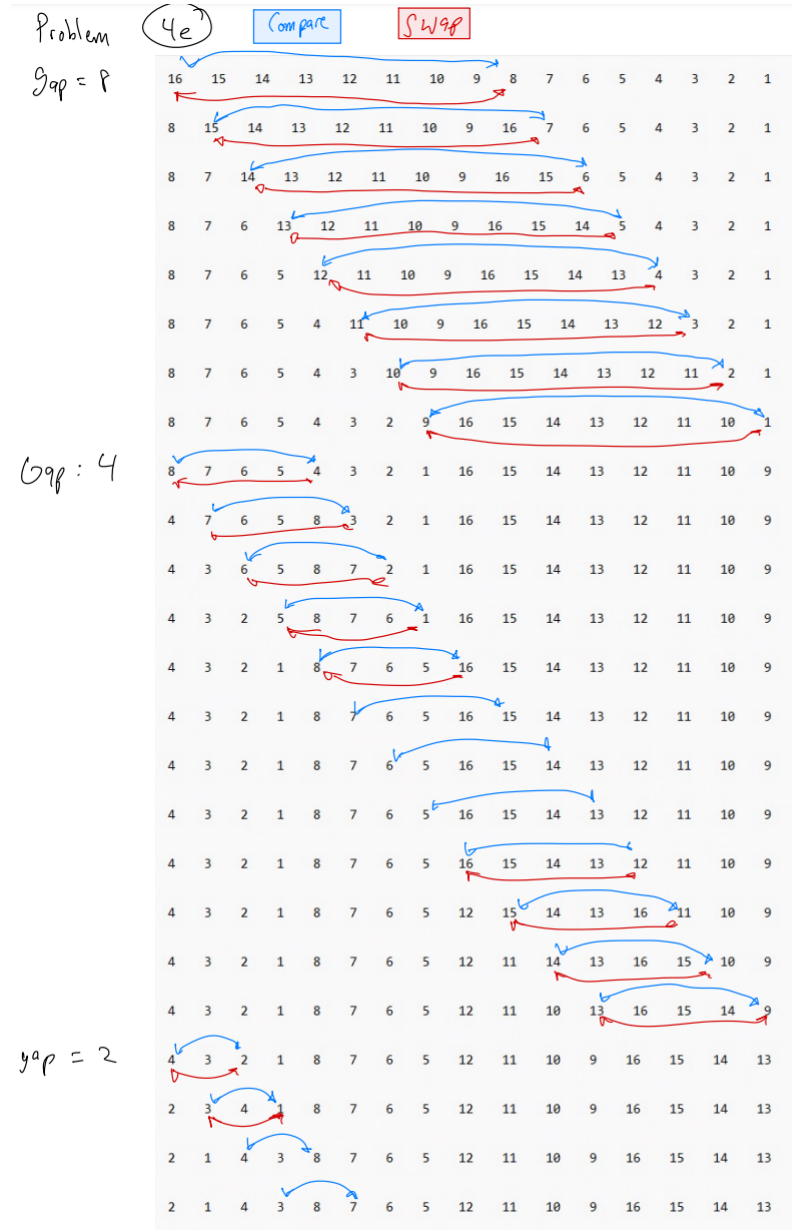
Second run - $n - 1$ compares, 1 swap.

The summation for compares:

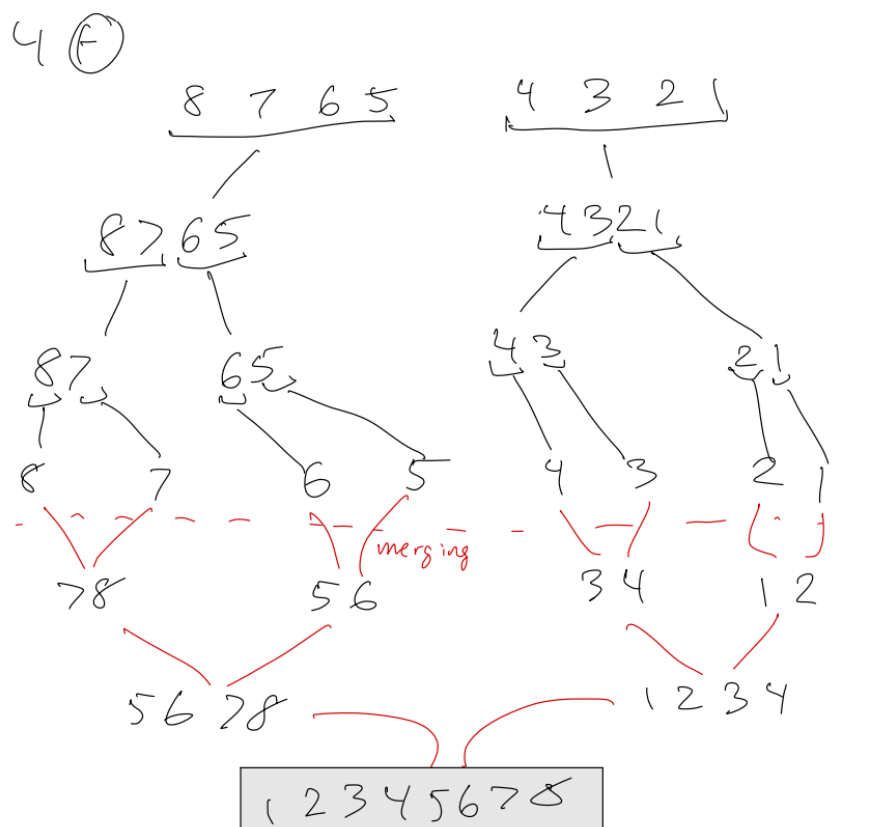
$$\begin{aligned}\sum_{i=1}^{n-1} i &= \frac{n(n-1)}{2} \\ &= \frac{n^2 - n}{2} \\ &= \frac{n^2}{2} - \frac{n}{2}\end{aligned}$$

The summation for swaps is just n , because every run only has one swap, the smallest element to its correct position.

(e) 49 compares, 33 swaps







(f)

Problem 5.

Resources used: None

Collaborators: None

(a) Inversion pairs:

(0, 4), (1, 4), (2, 3), (2, 4), (3, 4) - 5 pairs

(b) A reversed array has the most inversions. It has $n(n - 1)/2$ inversions.

(c) As inversion count increases, the runtime for insertion sort increases, Insertion sort is based on swapping a previous larger item with a smaller item that is after. The more inversions, the more swaps.

(d) Runtime analysis: It is the exact same as mergesort, because the only overhead that we are adding is memory related. The

Using the hint in the bottom of the question, I used the algorithm written in the slides and modified it for this purpose.

```
algorithm MergeSort(A, l, r)
    inversions = 0                                // this is the inversion counter
                                                // that will be incremented.
    if (l < r)
        m = (l + r) / 2
        inversions += MergeSort(A, l, m)
        inversions += MergeSort(A, m + 1, r)
        inversions += merge(A, l, m, r)
    end if

    return A, inversions
end algorithm
```

```
algorithm merge(A, l, m, r)
    n1 = m - l + 1
    n2 = r - m

    inv = 0 // counter for inversions
```

```
let L be an array of size n1 + 1
let R be an array of size n2 + 1

for (i = 0 to n1 - 1)
  L[i] = A[l + i]
end for

for (j = 0 to n2 - 1)
  R[j] = A[m + j + 1]
end for

L[n1] =  $\infty$ , R[n2] =  $\infty$ 
i = 0, j = 0

for (k = l to r - 1)
  if (L[i] <= R[j])
    A[k] = L[i]
    i = i + 1
  else
    A[k] = R[j]
    j = j + 1
    inv += 1
  end if
end for

return A, inv
end algorithm
```