Week 4

Searching & Sorting

Thorben Klabunde

www.th-kl.ch

13.10.2025

Agenda

1. Mini-Quiz
2. Assignment
3. Theory Recap
4. Additional Practice
5. Peer Grading

Mini-Quiz

Mini-Quiz 3

1. **Cl.:** Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be some functions. Then, we have $\max\{f(n), g(n)\} \leq O(f(n) + g(n))$.

   Answer: True. For any two positive numbers a, b, we know $\max\{a, b\} \leq a + b$. This holds for all $n \in \mathbb{N}$, so the claim is true.

2. **Cl.:** Suppose you have an algorithm with runtime $T(n)$ where $T(n) \leq T(n/2) + C$ for some constant $C > 0$ and $T(1) = 1$. Then, $T(n) \leq O(\log n)$.

   Answer: True. This is the recurrence for algorithms like Binary Search. By expansion, we get $T(n) \leq T(1) + C \log_2(n) = 1 + C \log_2(n)$, which is in $O(\log n)$.

3. **Cl.:** $n^n \geq \Omega(100^n)$.

   Answer: True. We check the limit $\lim_{n\to\infty} \frac{100^n}{n^n} = \lim_{n\to\infty} \left(\frac{100}{n}\right)^n = 0$. Since the limit is 0, $100^n = o(n^n)$, which implies $n^n = \Omega(100^n)$.

Mini-Quiz 3

4. **Cl.:** $n^5 + 10n^4 \log(n) = \Theta(n^5 \log(n))$.

   Answer: False. We check the limit $\lim_{n\to\infty} \frac{n^5+10n^4\log(n)}{n^5\log(n)} = \lim_{n\to\infty} \left(\frac{1}{\log(n)} + \frac{10}{n}\right) = 0 + 0 = 0$. Since the limit is 0, the numerator is o of the denominator, so $n^5 + 10n^4 \log(n) = o(n^5 \log(n))$, and therefore $O(n^5 \log(n))$ is false.

5. **Cl.:** $2^{\log_{10}(n)} = \Theta(2^{\ln(n)})$.

   Answer: False. We can write the right hand side as $2^{\log_{10}(n)} = 2^{\ln(n)/\ln(10)}$ so checking it by limits we have

$$\lim_{n\to\infty} \frac{2^{\ln(n)/\ln(10)}}{2^{\ln(n)}} = \lim_{n\to\infty} 2^{\ln(n)(1/\ln(10)-1)} = 0$$

   as $1/\ln(10) - 1$ is negative so the statement is false.

Mini-Quiz 3

6. Alice has designed a new comparison-based search algorithm for sorted arrays that works by iteratively squaring the search range starting at index 2. You have checked the algorithm and it correctly solves the problem. She claims that this algorithm runs in time $O(\log(\log(n)))$ in the worst case.

✓ **a.** Alice is mistaken, her algorithm is slower than $O(\log(\log(n)))$.
We know from the lecture that any comparison-based search algorithm requires $\Omega(\log n)$ time in the worst case, so a runtime of $O(\log(\log(n)))$ is not possible.

b. Yes, iterated squaring leads to a $O(\log(\log(n)))$ runtime.

c. Without knowing the details of the algorithm, we cannot make a statement about its worst-case runtime.

Mini-Quiz 3

Which sorting algorithm from the lecture does the pseudocode snippet below implement?

```
for j = 1, 2, ..., n-1 do:
    max = 1
    for k = 1, 2, ..., n-j+1 do:
        if A[k] > A[max] then:
            max = k
    if max < n-j+1 then:
        Swap A[max] and A[n-j+1]
```

✓ **a.** Selection Sort
b. Insertion Sort
c. Merge Sort
d. Bubble Sort

Mini-Quiz 3

6. Suppose you have an algorithm with runtime $T(n)$ on an input of size n. Assume you know that $T(n) \geq 2 \cdot T(n/2) + dn$ for some constant $d > 0$ and that $T(1) = 10$. Then, $T(n) \geq \Omega(n^{1.5})$.

   Answer: False. This recurrence resembles the runtime of Merge Sort, which we know is $O(n \log n)$. Since $\lim_{n \to \infty} \frac{n^{1.5}}{n \log n} = \infty$, we know that $n \log n$ is not in $\Omega(n^{1.5})$.

Mini-Quiz 3

9. **Cl.:** Linear search has a worst-case complexity of $\Theta(n)$ on sorted arrays.

   Answer: True. The array being sorted doesn't help linear search. In the worst case (e.g., searching for the last element), we still have to check all n elements.

Mini-Quiz 3

10. Consider the following pseudocode. Which expression describes the exact number of calls to $f()$?

```
i <- 1
while i <= n:
    j <- 1
    while j <= i:
        f()
        j <- j + 1
    i <- i + 1
```

✓ **a.** $\sum_{i=1}^{n} \left(\sum_{j=1}^{i} 1 \right)$

The outer loop runs from $i = 1$ to n. For each i, the inner loop runs from $j = 1$ to i, calling $f()$ once each time.

b. $\sum_{i=1}^{n} i \cdot \left(\sum_{j=1}^{i} 1 \right)$

c. $\sum_{i=1}^{n} \left(1 + \sum_{j=1}^{i} 1 \right)$

d. $\sum_{i=1}^{n} \left(\sum_{j=1}^{n} 1 \right)$

Assignment

Feedback Assignment 2

- **Fibonacci Exercise**: Remember to cover all required cases in the B.C. and, consequently, also in the I.H. At the latest, when writing your proof and using the I.H. you should check that all the assumed cases are covered.
- **Limits**: Generally well done but please double check your calculations for arithmetic errors. When you apply a Theorem, get in the habit of briefly stating it.
- **Bounds**: Well done! General feedback on proof structure: Try to keep your proofs clearly structured (left to right, top to bottom) and guide the reader on how your statements are connected.

Ex. 3.1.b (3)

(3) Prove that for $n \geq 3$

$$\frac{n^{1/n} - 1}{n} = \Theta\left(\frac{\ln(n)}{n^2}\right).$$

You may use results from the earlier exercises.

You may use the following fact:

Let $f, g : \mathbb{R}^+ \to \mathbb{R}$. Suppose that $\lim_{n\to\infty} g(n) = \infty$ and there exists some constant $C \in \mathbb{R}$ such that $\lim_{n\to\infty} f(n) = C$. Then $\lim_{n\to\infty} f(g(n)) = C$.

(3) $\dfrac{\frac{n^{1/n}-1}{n}}{\frac{\ln(n)}{n^2}} = \dfrac{n^{1/n}-1}{n}\cdot\dfrac{n^2}{\ln(n)} = \dfrac{n(n^{1/n}-1)}{\ln(n)} = \dfrac{n(e^{\ln(n^{1/n})}-1)}{\ln(n)} = \dfrac{n(e^{\frac{1}{n}\cdot\ln(n)}-1)}{\ln(n)}$

Let $f(n) = n(e^{1/n}-1)$ and $g(n) = \dfrac{n}{\ln(n)}$ and notice:

$\dfrac{n(e^{\frac{1}{n}\cdot\ln(n)}-1)}{\ln(n)} = f(g(n)) \overset{\text{(hint)}}{\Rightarrow} \lim_{n\to\infty} f(g(n)) = C \in \mathbb{R}$ since: $\begin{cases} \text{by (a): } \lim_{n\to\infty} g(n) = \infty \\ \text{by (b): } \lim_{n\to\infty} f(n) = C \in \mathbb{R} \end{cases}$

Ex. 3.2

Exercise 3.2 *Substring counting.*

Given a n-bit bitstring $S[0..n-1]$ (i.e. $S[i] \in \{0,1\}$ for $i = 0, 1, ..., n-1$), and an integer $k \geq 0$, we would like to count the number of nonempty contiguous substrings of S with exactly k ones. Assume $n \geq 2$.

For example, when $S =$ "0110" and $k = 2$, there are 4 such substrings: "011", "11", "110", and "0110".

Ex. 3.2c)

(c) Consider an integer $m \in \{0, 1, \ldots, n-2\}$. Using PREFIXTABLE and SUFFIXTABLE, design an algorithm SPANNING(m, k, S) that returns the number of substrings $S[i..j]$ of S that have exactly k ones and such that $i \leq m < j$.

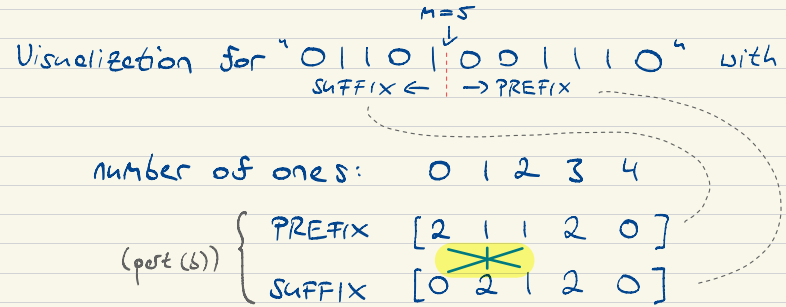
For example, if $S =$ "0110", $k = 2$, and $m = 0$, there exist exactly two such strings: "011" and "0110". Hence, SPANNING$(m, k, S) = 2$.

Describe the algorithm using pseudocode. Mention and justify the runtime of your algorithm (you don't need to provide a formal proof, but you should state your reasoning).

Hint: *Each substring $S[i..j]$ with $i \leq m < j$ can be obtained by concatenating a string $S[i..m]$ that is a suffix of $S[0..m]$ and a string $S[m+1..j]$ that is a prefix of $S[m+1..n-1]$.*

Algorithm 3

function SPANNING(m, k, S)
  $T_1 \leftarrow$ SUFFIXTABLE($S[0..m]$)
  $T_2 \leftarrow$ PREFIXTABLE($S[m+1..n-1]$)
  return $\sum_{p=\max(0,k-(n-m-1))}^{\min(k,m+1)} (T_1[p] \cdot T_2[k-p])$



Compute all combinations of prefix-sum + suffix-sum that yield k when combined.

Ex. 3.2d)

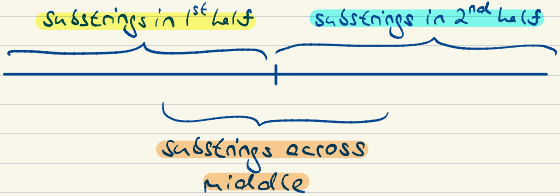
(d)* Using SPANNING, design an algorithm with a runtime[1] of at most $O(n \log n)$ that counts the number of nonempty substrings of a n-bit bitstring S with exactly k ones. (You can assume that n is a power of two.)

Justify its runtime. You don't need to provide a formal proof, but you should state your reasoning.

Hint: *Use the recursive idea from the lecture.*

Algorithm 4 Clever substring counting

function COUNTSUBSTR($S, k, i = 0, j = n - 1$)
  if $i = j$ **then**
    if $k = 1$ **and** $S[i] = 1$ **then**
      return 1
    else if $k = 0$ **and** $S[i] = 0$ **then**
      return 1
    else
      return 0
  else
    $m \leftarrow \lfloor (i + j)/2 \rfloor$
    return COUNTSUBSTR(S, k, i, m) + COUNTSUBSTR$(S, k, m + 1, j)$ + SPANNING$(m, k, S[i..j])$



Runtime: see (c)

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

$\Rightarrow T(n) \leq O(n \log n)$

(see last week's detailed analysis)

Ex. 3.4a)

(a) Design an $O(n)$ algorithm that computes the nth Fibonacci number f_n for $n \in \mathbb{N}$. Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).

Remark: As shown in part Exercise 2.2, f_n grows exponentially (e.g., at least as fast as $\Omega(1.5^n)$). For this exercise, you can assume that all addition operations can be performed in constant time.

(a)

Algorithm 7

$F[0..n] \leftarrow$ an array of $(n+1)$ integers } $O(1)$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2, \ldots, n$ **do** } $O(n)$
  $F[i] \leftarrow F[i-2] + F[i-1]$ } $O(1)$
return $F[n]$

There is a constant number of set-up operations and one for-loop performing n iterations, where each operation takes $O(1)$. Hence, in total we get $O(n)$.

Ex. 3.4b)

(b) Given an integer $k \geq 2$, design an algorithm that computes the largest Fibonacci number f_n such that $f_n \leq k$. The algorithm should have complexity $O(\log k)$. Describe the algorithm using pseudocode and formally prove its runtime is $O(\log k)$.

Hint: *Use the bound proved in Exercise 2.2.*

(b) Recall from last week's assignment that for the n^{th} Fibonacci number it holds that: $fib_n \geq \frac{1}{3} \cdot 1.5^n$

Notice then: $k \geq fib_n \geq \frac{1}{3} \cdot 1.5^n \Rightarrow \log_{1.5}(3 \cdot k) = \frac{\log(3) + \log(k)}{\log(1.5)} \geq n$

$\Rightarrow n \leq O(\log(k))$

given that we compute f_n in $O(n)$, we obtain a runtime of $O(\log(k))$

(we only need to execute $O(\log k)$ iterations of the loop)

Iterative Squaring:

Algorithm 10 Power(a, n)

```
if n = 1 then
    return a
else
    x ← Power(a, n/2)
    return x · x
```

Runtime: $T(n) = T(\frac{n}{2}) + 1$

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

$= \dots$

$= T(1) + \log n$

$\leq O(\log n)$

Theory Recap

Sorting Searching: The Big Picture

This week, the three key things you should learn are how to:

- **Characterize the Search & Sort Problem**
  Sorting data makes searching much faster ($O(n) \rightarrow O(\log n)$). Different sorting algorithms have fundamentally different runtimes ($O(n \log n)$ vs. $O(n^2)$).

- **Proving Theoretical Limits** to discuss Optimality
  Using **decision trees**, we proved a lower bound for an entire class of algorithms. For comparison-based search, this limit is $\Omega(\log n)$, making Binary Search optimal.

- **Prove Correctness using Loop Invariants**
  Loop invariants provide a formal way to prove iterative algorithms are correct using a simple inductive structure (base case, maintenance, termination).

Searching Algorithms at a Glance

The goal of a search algorithm is to find the index of an element b in an array A. The **efficiency** of the search **depends** heavily **on whether the array is sorted**.

Feature	**Linear Search**	**Binary Search**
Idea	Iterate through the array from the beginning until the element is found.	Compare the target with the middle element and discard half of the remaining array in each step.
Requirement	None. The array can be unsorted.	The array must be sorted.
Runtime	$\mathcal{O}(n)$ comparisons in the worst case.	$\mathcal{O}(\log n)$ comparisons.
Optimality	**Best possible** for an **unsorted** array.	Asymptotically **optimal** for **comparison-based** search.

Key Takeaway: Searching is significantly faster on sorted data.

Theoretical Lower Bound for Search: The Decision Tree Model

How can we prove that Binary Search, with its $\mathcal{O}(\log n)$ runtime, is the best we can do for searching in a sorted array?

Key Idea: The Decision Tree

Any comparison-based search algorithm can be modeled as a **decision tree**.

- Each **internal node** represents a comparison (e.g., $b < A[i]$?).
- Each **leaf node** represents a final outcome or result.

The worst-case number of comparisons for an algorithm is the height of its decision tree.

For an array A of size n, how many possible outcomes are there?

- The element b could be at any of the n indices.
- The element b might not be in the array at all.

This gives a total of $\mathbf{n + 1}$ possible outcomes.

Deriving the Lower Bound

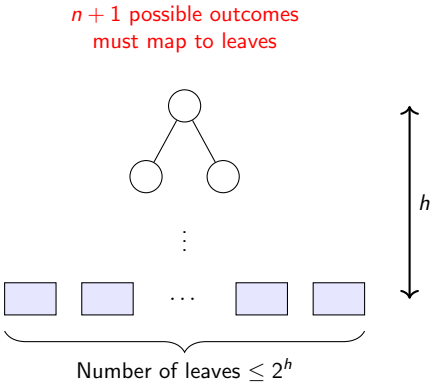
Theorem: Lower Bound for Comparison-Based Search

Any comparison-based search algorithm $\mathcal{A}$ on a sorted array of size n requires at least $\mathbf{\Omega(\log n)}$ comparisons in the worst case.

Proof Sketch:

1. Every path from root to leaf is an execution path. The tree's height is the worst-case number of comparisons.
2. $\mathcal{A}$ must distinguish between $\mathbf{n+1}$ possible outcomes, each a unique leaf.
3. A binary tree of height h has at most $\mathbf{2^h}$ leaves.
4. Since the number of leaves must be at least the number of outcomes:

$$n + 1 \leq 2^h \implies \log_2(n+1) \leq h$$



Bubble Sort

Idea: In each pass, let the largest unsorted element "bubble up" into its correct position.

Algorithm

```
BubbleSort(A)
    n = A.length;
    for j = 1 ... n-1 do:
        // inner loop can be
        // improved to n-j
        for i = 1 ... n-1 do:
            if A[i] > A[i+1] then
                swap A[i] and A[i+1];
            end if
        end for
    end for
```

Loop invariant (after j passes):

The largest j elements are in their final positions at the right end, i.e., the suffix $A[n-j+1..n]$ is sorted in asc. order.

Comparisons:

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

$\sum_{j=1}^{n-1}\sum_{i=1}^{n-j} 1 = \sum_{j=1}^{n-1}(n-j) = \sum_{j=1}^{n-1} j \overset{Gauss}{=} \Theta(n^2)$

Swaps:
$O(n^2)$ in the worst case.

Selection Sort

Idea: In each pass, find the largest unsorted element and swap it into its correct final position at the end of the array.

Algorithm

```
SelectionSort(A)
    int n = A.length;
    for j = 1 ... n-1:
        // note: max op. is also a for-loop
        k = idx of max(A[1...n-j]);
        swap A[k] and A[n-j]
    end for
```

Invariant $I(j)$

After j iterations of the outer loop, the last j elements of the array contain the j largest values in sorted order.

$A[1 \dots n-j]$	$A[n-j+1 \dots n]$
(unsorted)	(sorted)

Comparisons:

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

Swaps: $O(n)$ (only once per iteration)

Insertion Sort (with Binary Search for the Slot)

Idea: Build a sorted subarray at the beginning. Take the next element from the unsorted part and insert it into its correct place within the already sorted part.

Algorithm

```
insertionSort(A)
    int n = A.length;
    for j = 1 ... n-1:
    // bin. search for A[j+1] in A[1...j]
    // to find insertion idx as a by-product
        tmp = A[j+1]
        shift A[k...j] to A[k+1...j+1]
        A[k] = tmp
    end for
```

Invariant $I'(j)$

After j iterations of the outer loop, the prefix of the array $A[1 \dots j]$ is sorted.

$A[1 \dots j]$	$A[j+1 \dots n]$
(sorted)	(unsorted)

Comparisons:

$$\sum_{j=1}^{n-1} \lceil \log_2 j \rceil \leq O(n \log n)$$

Swaps: $\sum_{j=1}^{n-1} O(j) = \Theta(n^2)$ (shifts dominate).

Merge Sort

Idea: Divide and Conquer by splitting the array in half, sorting each half, and then merging the two sorted halves back together.

Algorithm

```
MergeSort(A, left, right)
    if left < right
        middle = (left + right) / 2
        MergeSort(A, left, middle)
        MergeSort(A, middle+1, right)
        Merge(A, left, middle, right)
    end if
```

Merge invariant:

At any time, B holds the sorted merge of the current prefixes of the two halves; after merging, $A[L..R]$ is sorted.

Time (comparisons/moves):

$T(n) = 2T(n/2) + cn \Rightarrow O(n \log n)$.

(*see last week's DnC Runtime Analysis*)

Extra space: $\Theta(n)$ for the auxiliary array during merge (MS does *not* act in-place)

Proving Correctness with Invariants

The analysis of an algorithm always includes a proof of correctness and runtime considerations.

One technique for iterative algorithms is to use a **loop invariant**, which allows us to make statements about the state of some property with every iteration.

Proof by Induction with Loop Invariants

We must show three things about our invariant:

1. **Initialization (Base Case):** The invariant is true before the first iteration of the loop.
2. **Maintenance (Inductive Step):** If the invariant is true before an iteration (our I.H.), it remains true after the iteration.
3. **Termination:** When the loop terminates, the invariant (combined with the loop's termination condition) guarantees the desired outcome.

Caution: The most critical part is choosing a precise and useful invariant!

Correctness of Bubble Sort

Let's **prove the correctness of Bubble Sort** using the invariant we discussed.

Bubble Sort Invariant

After j outer iterations, the suffix $A[n-j..n-1]$ contains the j largest elements in increasing order.

Your Turn: Proving Bubble Sort Correct

Invariant $I(j)$: After j outer iterations, the suffix $A[n-j..n-1]$ contains the j largest elements in increasing order.

Base Case: Let j = _ . The invariant holds because ...

I.H.: Assume now that ...

I.S.: Then in the $(j+1)$-st pass of the outer loop ...

Termination: The loop finishes after $j = n-1$ iterations. The whole array must be sorted since ...

Correctness of Bubble Sort

Goal: Upon termination, the array A is sorted in non-decreasing order.

Invariant $I(j)$: After j outer iterations, the suffix $A[n-j..n-1]$ contains the j largest elements of A in non-decreasing order.

B.C. ($j = 0$): Before the loop, the suffix $A[n..n-1]$ is empty, so the invariant holds trivially.

I.H.: Assume $I(j)$ holds for some $0 \leq j < n-1$.

I.S.: The $(j+1)$-th iteration scans the prefix $A[0..n-j-1]$ and moves its largest element to $A[n-j-1]$. By the I.H., $A[n-j..n-1]$ already contains the j largest elements, sorted. Therefore, the new suffix $A[n-(j+1)..n-1]$ now holds the $j+1$ largest elements in sorted order. Thus, $I(j+1)$ holds.

Termination: The loop terminates when $j = n-1$. By the invariant $I(n-1)$, the suffix $A[1..n-1]$ contains the $n-1$ largest elements, sorted. Consequently, the remaining element $A[0]$ must be the smallest. The entire array A is therefore sorted.

Questions?

Additional Practice

Counting Loop Iterations - FS21

Algorithm 2

```
for j = 1, ..., n do
    for k = 1, ..., n do
        ℓ ← 1
        while k + ℓ ≤ j do
            f()
            ℓ ← 2 · ℓ
```

⇒

```
for j = 1 ... n do
    for k = 1 ... j-1        (adjust index range to ensure log is defined)
        t ← 0
        while t ≤ log(j-k)   (*)
            f()
            t ← t+1
```

Notice that we can easily turn the outer two loops into sums.
We also want to express the inner loop as a sum but notice that:
(i) the new variable ℓ is an exponential of 2; (ii) the loop condition isn't immediately clear.

To solve (i), notice that $\ell = 2^t$ for some $t \in \mathbb{N}$. To linearize the loop variable, we apply the log and have $t = \log(\ell)$.

(*) Notice that for the log to be defined we only iterate k up to $j-1$ otherwise the log is not defined.

Then we reformulate: $k+\ell \le j \Leftrightarrow k+2^t \le j \Leftrightarrow 2^t \le j-k \Leftrightarrow t \le \lfloor \log(j-k) \rfloor$

$$\Rightarrow A(n) = \sum_{j=1}^{n} \sum_{k=1}^{j-1} \sum_{t=0}^{\lfloor \log(j-k) \rfloor} 1 = \sum_{j=1}^{n} \sum_{k=1}^{j-1} \lfloor \log(j-k) \rfloor + 1$$

(reverse the order of summ.)

reversed order of summ.

$$= \sum_{j=1}^{n} \underbrace{\sum_{k=1}^{j-1} \lfloor \log(k) \rfloor + 1}_{\Theta(j \log j)} \overset{(I)}{=} \sum_{j=1}^{n} \Theta(j \log j) \overset{(II)}{=} \underline{\Theta(n^2 \log n)} \quad \square$$

We again use the technique of splitting the sums in half:

(I) $\sum_{k=\lceil \frac{j-1}{2} \rceil}^{j-1} \log(\frac{j-1}{2}) \le \sum_{k=1}^{j-1} \lfloor \log(k) \rfloor \le \sum_{k=1}^{j-1} \log(j)$

$\Theta(j(\log(j))$ $\qquad$ $\Theta(j \log j)$

(II) $\sum_{j=\lceil \frac{n}{2} \rceil}^{n} \frac{n}{2} \log \frac{n}{2} \le \sum_{j=1}^{n} \Theta(j(\log j) \le n^2 \log n$

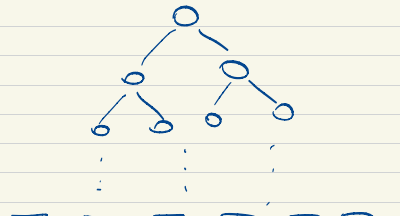
$\Theta(n^2 \log n)$ $\qquad$ $\Theta(n^2 \lg n)$

Theoretical Lower Bound for Sorting

Prove that the theoretical lower bound for any **comparison-based** sorting algorithm $\mathcal{A}$ is $\Omega(n \log n)$.

Proof.

Hint: Proceed analogously to the proof for search. Characterize the algorithm as a binary decision trees, count the leaves and conclude the result.



$\mathcal{A}$ must distinguish between $n!$ diff. sortings

$\Rightarrow$ there are at least $n!$ leaves in the tree

Recall that: # leaves $\leq 2^h$ (a tree of height h has at most 2^h leaves)

$\Rightarrow \quad h \geq \log(\#\text{leaves}) \geq \log(n!)$

$\Rightarrow \quad h \geq \Omega(n \log n)$

where the final inequality follows from

$$\log(n!) \overset{(1)}{\geq} \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right) = \Theta(n \lg n)$$

(1) same trick as before: $\underbrace{n(n-1)\cdot\ldots\cdot\left(\frac{n}{2}\right)}_{\geq \left(\frac{n}{2}\right)^{\frac{n}{2}}} \cdot \underbrace{\left(\frac{n}{2}-1\right)\cdots(1)}_{\geq 1}$

Quiz: Searching and Sorting

Claim	True	False
1. Bubble Sort algorithm always performs $\mathcal{O}(n^2)$ comparisons.	☐	☐
2. The invariant for Selection Sort is that after j passes, the first j elements are sorted.	☐	☐
3. In the worst case, Selection Sort performs $\mathcal{O}(n)$ swaps.	☐	☐
4. Linear search can take $\mathcal{O}(\log n)$ operations on some arrays.	☐	☐

ii) Assume $n \geq 10$. For which of the following arrays does `InsertionSort` take time $\Omega(n^2)$ (to sort the array in ascending order)?

(a) $[2, 1, 3, 4, 5, \ldots, n-2, n-1, n]$

(b) $[n, n-1, n-2, \ldots, 3, 2, 1]$

(c) For both

(d) For neither

Peer Grading

Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 3.1**

Each group grades the group below in the table I sent you (resp. the last one grades the first one). Please send the other group your solution. If you don't get their solution, please contact me so I can send it to you.