Computer Science Fundamentals:
Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

We start at high-level understandings (admittedly somewhat dry with definitions), gaining familiarity with data-structures and algorithms (where the fun begins), number systems (e.g,. binary), which will allow us to dive deeper and create our own theoretical models of computation, i.e., a computer (plenty visuals, less text). After such, we revisit algorithms and data-structures, learning the classic methods and strategies that motivated our modern system today (The must knows for any interview or real-world application).

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
Dora Erdos and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms,
with contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne,
at Boston University

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

You can skip this: These prerequisites help, but they are not strictly necessary. Don't spend much if any time here. It may serve as a reference later on. Most if not all definitions are self contained and don't skip any logical steps.

However it would be helpful (not strictly) to be familiar with at least one programming language, specifically **Java** as we reference the language a few times. Not that we'll need to code, or will use them, but easily be able to read pseudocode (make-believe code that often strongly resembles real programming languages).

Use this entire text as a supplementary resource—If it doesn't make sense, cross reference other sources,and hopefully comeback and improve this one, as it's open-source.

Theorem 0.1: Common Derivatives

Power Rule: For $n \neq 0$	$\frac{d}{dx}(x^n) = n \cdot x^{n-1}$. E.g., $\frac{d}{dx}(x^2) = 2x$
Derivative of a Constant:	$\frac{d}{dx}(c) = 0$. E.g., $\frac{d}{dx}(5) = 0$
Derivative of ln:	$\frac{d}{dx}(\ln x) = \frac{1}{x}$
Derivative of $\log_a$:	$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$
Derivative of $\sqrt{x}$:	$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$
Derivative of function $f(x)$:	$\frac{d}{dx}(x) = 1$. E.g., $\frac{d}{dx}(5x) = 5$
Derivative of the Exponential Function:	$\frac{d}{dx}(e^x) = e^x$

Theorem 0.2: L'Hopital's Rule

Let $f(x)$ and $g(x)$ be two functions. If $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$, or $\lim_{x\to a} f(x) = \pm\infty$ and $\lim_{x\to a} g(x) = \pm\infty$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Where $f'(x)$ and $g'(x)$ are the derivatives of $f(x)$ and $g(x)$ respectively.

Theorem 0.3: Exponents Rules

For $a, b, x \in \mathbb{R}$, we have:

$$x^a \cdot x^b = x^{a+b} \text{ and } (x^a)^b = x^{ab}$$

$$x^a \cdot y^a = (xy)^a \text{ and } \frac{x^a}{y^a} = \left(\frac{x}{y}\right)^a$$

Note: The := symbol is short for "is defined as." For example, $x := y$ means x is defined as y.

Definition 0.1: Logarithm

Let $a, x \in \mathbb{R}$, $a > 0$, $a \neq 1$. Logarithm x base a is denoted as $\log_a(x)$, and is defined as:

$$\log_a(x) = y \iff a^y = x$$

Meaning *log* is inverse of the exponential function, i.e., $\log_a(x) := (a^y)^{-1}$.

Tip: To remember the order $log_a(x) = a^y$, think, "base a," as a is the base of our *log* and y.

Theorem 0.4: Logarithm Rules

For $a, b, x \in \mathbb{R}$, we have:

$$\log_a(x) + \log_a(y) = \log_a(xy) \text{ and } \log_a(x) - \log_a(y) = \log_a\left(\frac{x}{y}\right)$$

$$\log_a(x^b) = b\log_a(x) \text{ and } \log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

Definition 0.2: Permutations

Let $n \in \mathbb{Z}^+$. Then the number of distinct ways to arrange n objects in order is $n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. When we choose r objects from n objects, it's Denoted:

$$^{n}P_{r} := \frac{n!}{(n-r)!}$$

Where $P(n, r)$ is read as "n permute r."

Definition 0.3: Combinations

Let n and k be positive integers. Where order doesn't matter, the number of distinct ways to choose k objects from n objects is it's *combination.* Denoted:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Where $\binom{n}{k}$ is read as "n choose k.", and $\binom{\cdot}{\cdot}$, the *binomial coefficient.*

Theorem 0.5: Binomial Theorem

Let a and b be real numbers, and n a non-negative integer. The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

which expands explicitly as:

$$(a+b)^n = \binom{n}{0} a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \cdots + \binom{n}{n-1} a b^{n-1} + \binom{n}{n} b^n$$

where $\binom{n}{k}$ represents the binomial coefficient, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $0 \leq k \leq n$.

Theorem 0.6: Binomial Expansion of 2^n

For any non-negative integer n, the following identity holds:

$$2^n = \sum_{i=0}^{n} \binom{n}{i} = (1+1)^n.$$

Definition 0.4: Well-Ordering Principle

Every non-empty set of positive integers has a least element.

Definition 0.5: "Without Loss of Generality"

A phrase that indicates that the proceeding logic also applies to the other cases. i.e., For a proposition not to lose the assumption that it works other ways as well.

Theorem 0.7: Pigeon Hole Principle

Let $n, m \in \mathbb{Z}^+$ with $n < m$. Then if we distribute m pigeons into n pigeonholes, there must be at least one pigeonhole with more than one pigeon.

Theorem 0.8: Growth Rate Comparisons

Let n be a positive integer. The following inequalities show the growth rate of some common functions in increasing order:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

These inequalities indicate that as n grows larger, each function on the right-hand side grows faster than the ones to its left.

— 1 —

Scheduling

1.1 Interval Scheduling

Scheduling problems arise in many areas, such as scheduling classes, tasks, or jobs. Interval scheduling is a type of scheduling problem where we want to maximize the number of tasks we can complete.

Definition 1.1: Schedule

A **schedule** is a set of tasks which we call **jobs**. Each job has a start time s_i and an end time f_i. Two jobs are **compatible** if they do not overlap.

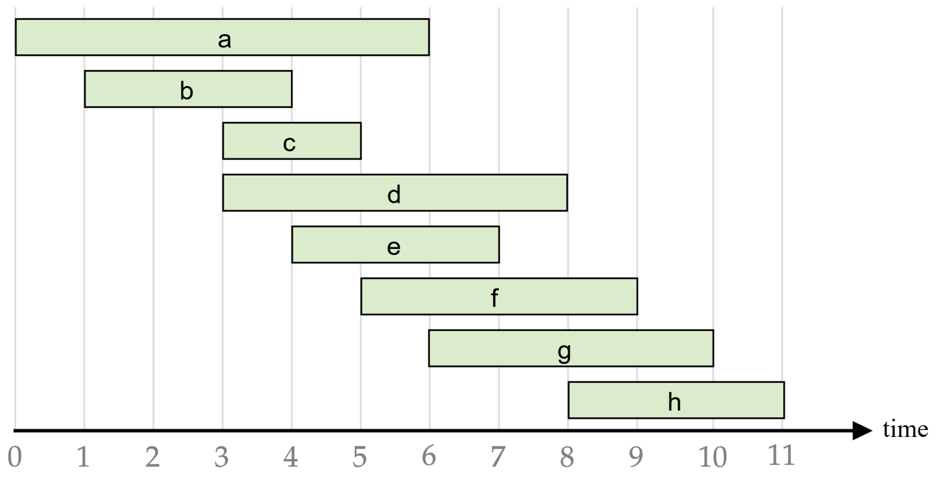

Figure 1.1: Given jobs a through h we find the largest subset of mutually compatible jobs $\{b, e, h\}$.

Definition 1.2: Greedy Algorithm

A **greedy algorithm** is an algorithm that makes the best choice at each step. I.e., it cares not about the future or big picture, only the immediate benefit, for fast computations.

Possible Approaches: Let s_j and f_j be the start and finish times of job j.

- [**Earliest Start Time**]**:** Consider jobs in ascending order of s_j.
- [**Earliest Finish Time**]**:** Consider jobs in ascending order of f_j.
- [**Shortest Interval**]**:** Consider jobs in ascending order of $f_j - s_j$.
- [**Fewest Conflicts**]**:** For each j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

We choose the **Earliest Finish Time** approach:

Proof 1.1: Greedy Algorithm Earliest Finish Time Correctness

Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by the greedy algorithm.
Let $j_1, j_2, \ldots, j_m$ denote the set of jobs in an optimal solution, with

$$i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r \text{ for the largest possible value of } r.$$

We can swap j_{r+1} for i_{r+1} in the optimal schedule, and it will still remain compatible. We repeat swaps until $r = k$. It's not possible that $m > k$ because j_{k+1} is compatible with i_k. ∎

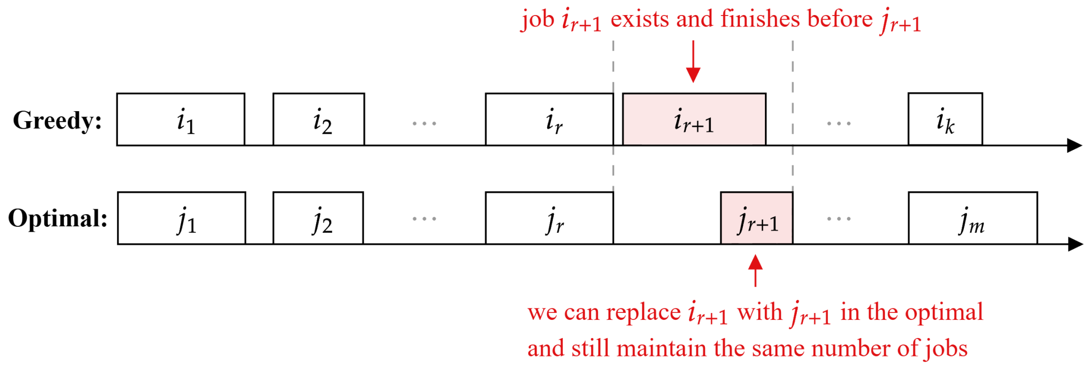
job i_{r+1} exists and finishes before j_{r+1}
Greedy:
i_1
i_2
$\cdots$
i_r
i_{r+1}
$\cdots$
i_k
Optimal:
j_1
j_2
$\cdots$
j_r
j_{r+1}
$\cdots$
j_m
we can replace i_{r+1} with j_{r+1} in the optimal
and still maintain the same number of jobs

Figure 1.2: At the first divergence, i_{r+1}, it is interchangeable with j_{r+1} (the optimal solution's choice); As if we were to cut the function f off at this moment, $f_{i_{r+1}} \leq f_{j_{r+1}}$. Hence, by induction, if we were to keep doing this, our greedy solution remains optimal (same number of jobs).

Theorem 1.1: Interval Scheduling & Earliest Finish Time

Given a set of jobs j with start and finish times s_j and f_j, we can obtain an optimal like solution by scheduling in ascending order of f_j, choosing the next compatible job.

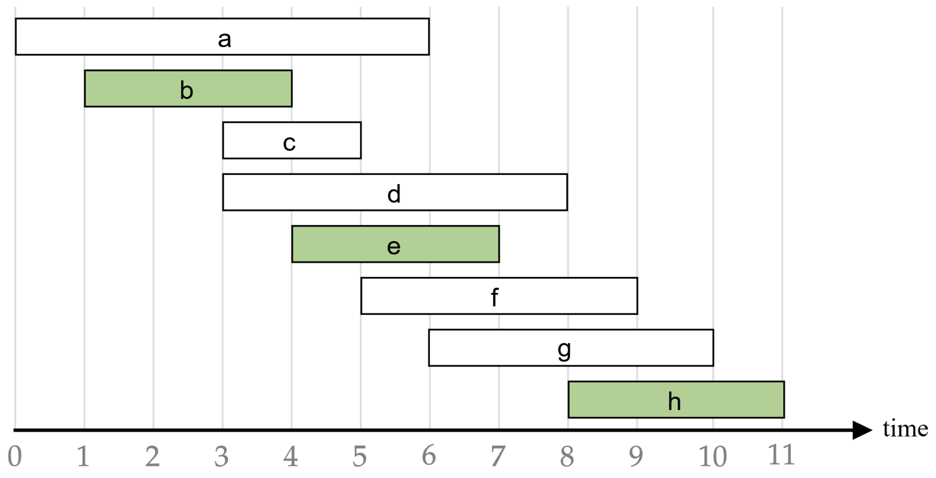

Figure 1.3: Solution to Figure (1.1) using early finish time first, yielding $\{b, e, h\}$.

Function 1.1: EarliestFinishTimeFirst Algorithm - `EFT(`$s_1, \ldots, s_n, f_1, \ldots, f_n$`)`

Finds the maximum set of non-overlapping jobs based on earliest finish time.

Input: A set of jobs with start times s_j and finish times f_j.
Output: The maximum set of selected jobs.

```
1 sorted_jobs ← sort($f_1, \ldots, f_n$) // sort by finish time $S \leftarrow \emptyset$ // selected jobs
    $f_{\text{last}} \leftarrow -\infty$;
2 for each $j$ in sorted_jobs do
3     if $f_{last} \leq s_j$ then
4         $S \leftarrow S \cup \{j\}$;
5         $f_{\text{last}} \leftarrow f_j$;
6 return $S$
```

Time Complexity: $O(n \log n)$ assuming our sorting algorithm is $O(n \log n)$. Then we iterate through n jobs.
Space Complexity: $O(n)$ storing the input of n jobs.

1.2 Interval Partitioning

Interval partitioning generalizes our interval scheduling to multiple resources, allowing them to run in parallel.

Definition 2.1: Interval Partitioning

Given a schedule of jobs j with start and finish times s_j and f_j. We **partition** jobs into a minimal amount of k resources such that no two jobs on the same resource overlap.

Scenario: ***Class Scheduling***
Say we have n classes and k classrooms. What are the minimum number of classrooms needed to schedule all classes?

Example: Let $n = \{a, b, c, \dots, i\}$ be classes with start and finish times. We attempt to find the minimum number of k classrooms needed to schedule all classes.

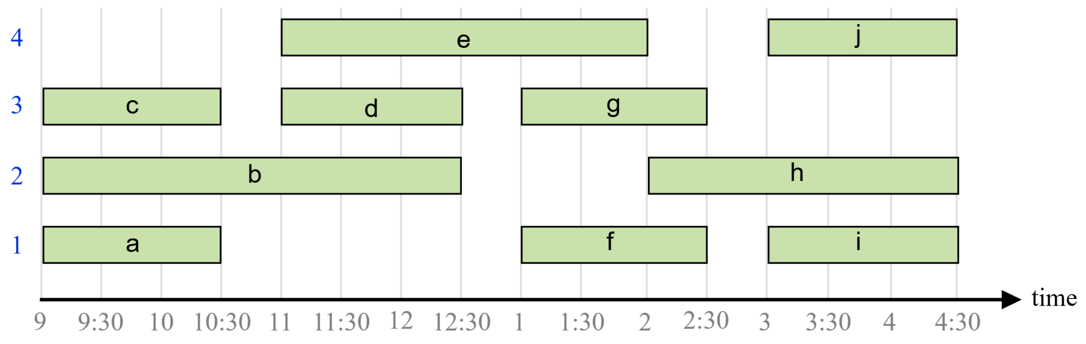

Figure 1.4: Though not optimal, here is a possible schedule where $k = 4$.

We strategies and figure out the minimum number of classrooms needed to schedule all classes in the worst-case. We observe in Example (1.2) that $\{c, b, a\}$ strictly overlap. Moreover, there are at most 3 classes overlapping at any time. Thus, we need at least 3 classrooms.

Theorem 2.1: Minimality of Interval Partitioning

Given a set of jobs j, c conflicting tasks, and k resources. We find the optimal k by $k = \max(c)$.

Possible Approaches: Let s_j and f_j be the start and finish times of job j.

- [**Earliest Start Time**]**:** Consider jobs in ascending order of s_j.
- [**Earliest Finish Time**]**:** Consider jobs in ascending order of f_j.
- [**Shortest Interval**]**:** Consider jobs in ascending order of $f_j - s_j$.
- [**Fewest Conflicts**]**:** For each j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Theorem 2.2: Interval Partitioning & Earliest Start Time

Given a set of jobs j with start and finish times s_j and f_j, we can obtain an optimal like solution by scheduling in ascending order of s_j. If two jobs overlap, we allocate a new resource.

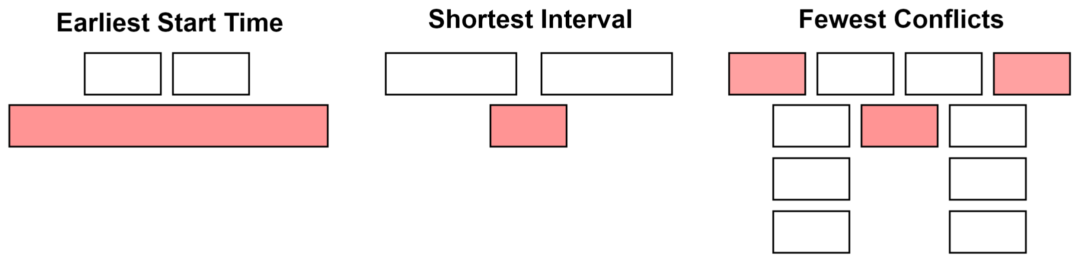

Figure 1.5: Counter examples of Earliest Start Time, Shortest Interval, and Fewest Conflicts. Earliest time fails to see the later two smaller jobs, shortest interval fails to see two longer jobs, fewest conflicts fails to see the sequential solution of 4 jobs because it has too many conflicts.

Proof 2.1: Classroom Allocation by Early Start Time First

Let d be the number of classrooms that the algorithm allocates:

(i.) Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d-1$ other classrooms.

(ii.) These d lectures each end after s_j.

(iii.) Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.

All schedules use $\geq d$ classrooms. Thus, we have d lectures overlapping at time $s_j + \epsilon$.

■

Function 2.1: EarliestStartTimeFirst Algorithm - EST($j = 1 \dots n : s_j, f_j$)

Finds an optimal schedule of lectures based on their earliest start time.

Input: A set of lectures with start times s_j and finish times f_j.
Output: Assignment of lectures to rooms.

```
1 $\mathcal{A} \leftarrow$ empty hash table      // $\mathcal{A}[k]$ contains the list of lectures assigned to
    room $k$ sorted_class $\leftarrow$ sort($s_1, \dots, s_n$)          // sort lectures by start time
2 for each $c$ in sorted_class do
3     $k \leftarrow$ find_compatible_room($c$, $\mathcal{A}$, $Q$);
4     if $k$ is not None then
5         $\mathcal{A}[k]$.add($c$);
6     else
7         $d \leftarrow$ len($\mathcal{A}$)      // highest room id $\mathcal{A}[d+1] \leftarrow$ [ ]      // open new room
           $\mathcal{A}[d+1]$.add($c$);
8 return $\mathcal{A}$
```

Time Complexity: $O(n \log n)$ assuming our sorting algorithm is $O(n \log n)$. Then we iterate through n jobs.
Space Complexity: $O(n)$ storing the input of n jobs.

1.3 Priority Queues: Min & Max Heaps

In this section we will discuss **min-heaps** which will help us sort maintain elements in a sorted data structure.

Definition 3.1: Balanced Tree

A **balanced tree** is a tree where the height of the left and right subtrees of every node differ by at most one.

Tip: To spot this, observe each level of the tree and see whether after consecutive levels, one branch has more nodes than the other.

Below is an example of a balanced tree and an unbalanced tree:

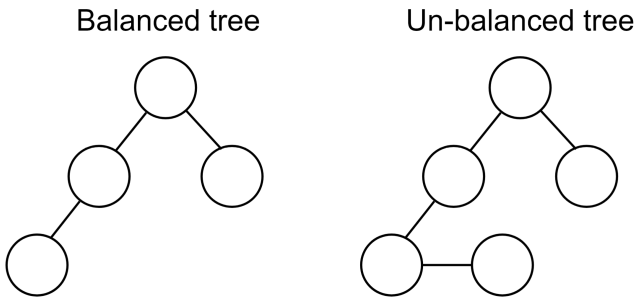

Figure 1.6: Examples of a balanced tree and an unbalanced tree

Definition 3.2: Binary Search Tree

A **binary tree** is a where each parent node p_i has at most two children c_{left} and c_{right}.
A **binary search tree** has each child ($c_{left} >$ p$_i$) and ($c_{right} <$ p$_i$).

With n nodes, there are $n-1$ edges.

Definition 3.3: Heap Tree

A **heap** (not to be confused with a memory heap) is a binary tree with the following properties:

(i.) It is a **complete binary tree** (a binary and balanced tree).

(ii.) It is a **min-heap** if the parent node is less than its children.

(iii.) It is a **max-heap** if the parent node is greater than its children.

Operations: Suppose we have a heap of n elements:

- **PEEK:** Return the root. $O(1)$;
- **INSERT:** Add a new element. $log(n)$;
- **EXTRACT:** Remove the root. $log(n)$;
- **UPDATE:** Update an element. $log(n)$;

We may also represent a heap as an array:

Definition 3.4: Heap as Array

heap can be represented as an array A where:

- The root is at index 0.
- The left child of node i is at index $2i + 1$.
- The right child of node i is at index $2i + 2$.

This representation allows us to efficiently access parent and child nodes, instead of using whole node objects.

Tip: The difference between a min-heap and a binary search tree, is that $c_{left} \leq c_{right} \leq \mathrm{p}_i$. That is, the left and right child in a min-heap are not ordered, just less than the parent; Contrary to a binary search tree where the left child is less than the parent and the right child is greater.

We use a min-heap as a data structure to maintain a sorted list as an input.

Function 3.1: find_compatible_room - `FCR(`c`,` A`,` Q`)`

Finds a compatible room for class c based on the current room schedule.

Input: Class ID c, current schedule A, priority queue Q with room finish times.
Output: The room k compatible with class c or `None`.

```
// $c$: class id, $A$: current schedule of room assignments
// $Q$: priority queue with room finish times
$\langle f_k, k \rangle \leftarrow$ PEEK_MIN($Q$)        // shows lowest ⟨key, value⟩ pair, $O(1)$
if $s_c > f_k$                                 // finish time in room $k$ then
    return $k$                                 // $c$ is compatible with room $k$
else
    return None;
```

Time Complexity: $O(n)$ as now we only need to check the minimum finish time in the priority queue.
Space Complexity: $O(n + m)$ if our min-heap is implemented as a hash table.

We can also implement a min-heap for sorting classes as well.

Function 3.2: EarliestStartTimeFirst Algorithm - EST($j = 1 \dots n : s_j, f_j$)

Finds an optimal schedule of lectures based on their earliest start time.

Input: Start times s_j and finish times f_j of classes.
Output: Assignment of lectures to rooms.

```
   // $s_j, f_j$: start and finish times of classes
 1 $\mathcal{A} \leftarrow$ empty hash table        // $\mathcal{A}[k]$ contains the list of courses assigned to
     room $k$ $Q \leftarrow$ empty priority queue      // contains ⟨finishTime, roomId⟩ pairs
     sorted_class $\leftarrow$ sort($s_1, \dots, s_n$)                    // sort by start time
 2 for each c in sorted_class do
 3     $k \leftarrow$ find_compatible_room($c$, $\mathcal{A}$, $Q$);
 4     if $k$ is not None then
 5         $\mathcal{A}[k]$.add($c$);
 6         $Q$.UPDATE-KEY($\langle f_k, k\rangle, \langle f_c, k\rangle$)        // update finish time of room $k$
 7     else
 8         $d \leftarrow$ len($\mathcal{A}$)     // highest room id $\mathcal{A}[d+1] \leftarrow [\,]$      // open new room
             $\mathcal{A}[d+1]$.add($c$);
 9         $Q$.INSERT($\langle f_c, d+1\rangle$);
10 return $\mathcal{A}$
```

Time Complexity: $O(n \log n)$ as inserting into a min heap is $O(\log n)$ and reading is $O(1)$.
Space Complexity: $O(n)$ storing the input of n classes.

Theorem 3.1: Heap Array Representation

A heap H can be represented by a zero-indexed array A via:

(i.) The root is at index 0.

(ii.) The left child of node i is at index $2i + 1$.

(iii.) The right child of node i is at index $2i + 2$.

Enabling a **space complexity** of $O(n)$.

1.4 Minimizing Lateness

Scenario: *Swamped in Assignments*
Say we have j assignments, each j assignment takes p_j time to prepare, and a d_j deadline. We want to finish all assignments, but minimize the overall lateness.

$j =$	1	2	3	4	5
p_j	3	2	1	4	2
d_j	4	7	8	8	10

Table 1.1: Table showing d_j and p_j deadlines and prepare times for j assignments.

Possible Approaches: Let s_j and f_j be the start and finish times of job j.

- [**Shortest Processing Time**]**:** shortest processing time p_j first.
- [**Earliest Deadline**]**:** earliest due time d_j first.
- [**Least Slack Time**]**:** least slack time $d_j - p_j$ first.

Counter Examples: Consider the following:

Shortest processing time p_j first:

	p_j	d_j
1	1	100
2	10	10

Table 1.2: If shortest p is picked, 1 will be placed first, despite its deadline being 100. Then 2 will be placed, which will be late by 1, since p_1+ $p_2 = 11$, missing its 10 deadline.

Smallest slack $d_j - p_j$ first:

	p_j	d_j
1	1	2
2	4	4

Table 1.3: 4−4 would have the smallest slack, so it's placed first, but this means 1 will be late by 3, since $4 + 1 = 5$ and $d_1 = 2$. Vice-versa, if 1 is first, then 2 is only late by 1.

Theorem 4.1: Minimizing Lateness

Given a set of jobs j with prepare and deadline times p_j and d_j under one resource, a like-optimal solution is obtained by scheduling deadlines in ascending order.

Tip: Video version of this section: https://youtu.be/wdqkJ7VICkc?t=973

Definition 4.1: Inversion

An **inversion** is a pair of jobs i and j such that $d_i > d_j$ but i is scheduled before j.

Proof 4.1: Minimizing Lateness by Earliest Deadline First

Let S' be an optimal schedule (such a schedule exists through brute force). If S' has no inversions, then $S' = S$ by definition of the greedy schedule.

Let there be S' with inversions, and sequentially eliminate them, by swapping adjacent inverted jobs. After

$$\leq \binom{n}{2} = O(n^2)$$

swaps, where n is the number of jobs, S' has no inversions and hence is identical to S.

Now, let us examine a single inversion i and j who both finish before time f_i; Swapping them does not increase the lateness of either job. Say i takes x time and j takes y time. Before the swap the time is $t + x + y$, where t is the time before i starts. After the swap the time is $t + y + x$. Thus, the lateness of both jobs remains unchanged. Then by induction, the next pair should also hold, meaning S retains an optimal schedule. ■

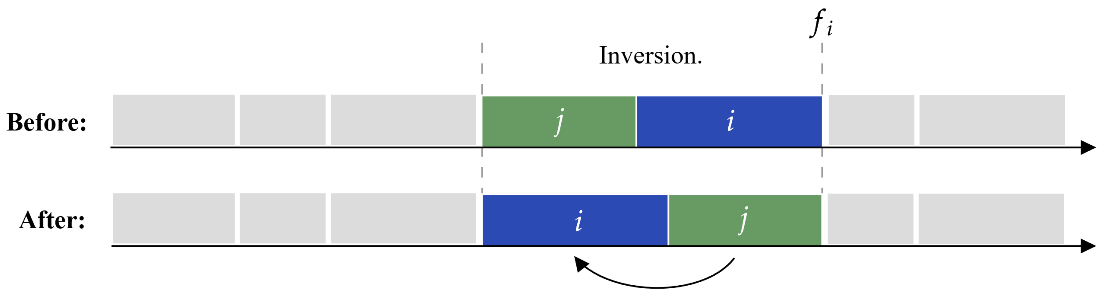

Figure 1.7: Shows an optimal solution with an inversion before and after a swap. Though j's deadline was satisfied before the swap ending at time f_i; after the swap it remains satisfied.

Tip: Say you have to clean your room completely by ℓ time. You are confused whether to clean your bed first or your desk. The desk takes d time and the bed takes b time. Whether you choose to clean your bed or your desk first, does not make a difference, as $d + b$ will always be the same.

Function 4.1: EarliestDeadlineFirst Algorithm - EDF($j = 1 \dots n : t_j, d_j$)

Schedule jobs based on their earliest deadline first.

Input: Length and deadlines of jobs.
Output: Intervals assigned to each job.

```
   // length and deadline of jobs
1  sorted ← sort($d_1, d_2, \dots, d_n$)  // sort by increasing deadline intervals ← empty list;
2  $t \leftarrow 0$                                        // keep track of time
3  for each $j$ in sorted do
       // assign job $j$ to interval $[t, t + t_j]$
4      intervals.add($[t,\ t + t_j]$);
5      $t \leftarrow t + t_j$;
6  return intervals
```

Time Complexity: $O(n \log n)$ assuming our sorting algorithm is $O(n \log n)$. Then we iterate through n jobs.
Space Complexity: $O(n)$ storing the input of n jobs, and we maintain an array of our intervals. $n + n = 2n = O(n)$.

Applying this algorithm back to Figure (1.1), we get the following optimal schedule:

$j =$	1	2	3	4	5
p_j	3	2	1	4	2
d_j	4	7	8	8	10

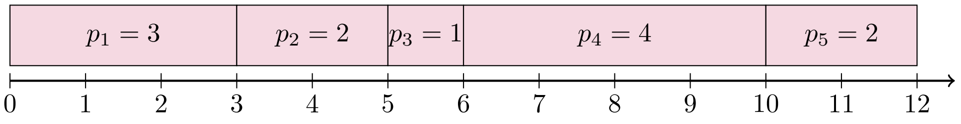

Figure 1.8: Sorting by earliest deadline first, we get the optimal schedule. Here, p_4 and p_5 only have a 2, totalling to 4 lateness.

Bibliography