Checklist before Submitting

In order to protect yourself against an unnecessary loss of points, use the following checklist before submitting:

- ☐ Crosscheck your report against the assignment instructions.
- ☐ Crosscheck against the technical writing and LaTeX advice below. The *English Style Guide* of UU at `https://mp.uu.se/en/web/info/stod/kommunikation-riktlinjer/sprak/eng-skrivregler` and the technical-writing *Checklist & Style Manual* of the Optimisation group at `http://optimisation.research.it.uu.se/checkList.pdf` offer many further pieces of advice. Common errors in English usage are discussed at `https://brians.wsu.edu/common-errors`. In particular, common errors in English usage by native Swedish speakers are listed at `http://www.crisluengo.net/index.php/english-language`.
- ☐ Spellcheck all documents, including the comments in the source code.
- ☐ Proofread, if not grammar-check, your report at least once per teammate.
- ☐ Crosscheck your source code against the coding convention.
- ☐ Submit your pdf report and your source code files in studium.

  Remember that when submitting you implicitly certify (a) that your report and all its uploaded attachments were produced solely by your team, except where explicitly stated otherwise and clearly referenced, (b) that each teammate can individually explain any part starting from the moment of submitting your report, and (c) that your report and attachments are not freely accessible on a public repository.

Python Coding convention

Function Specification

Every function must fulfil its **specification**:

- the function *must* contain a function header with the **type hints** of the arguments and results,
- the function *must* contain a comment with the **pre-condition** on the arguments, by referring to their names; write "`(none)`" instead of writing nothing if there is no pre-condition,
- the function *must* contain a comment with the **post-condition** on the arguments and results, by referring to all their names; an empty or "`(none)`" post-condition only makes sense if every result of the function is correct, and
- optionally the function *may* contain illuminating **examples** and **counter-examples** of argument-result pairs.

Recursion Invariant

Additionally, every recursive function *must* be commented with the **recursion variant**: a quantity that provably strictly decreases at every recursion according to some well-founded order towards some constant lower bound, thereby establishing **finite** execution of the recursion.

Loops

You *may* comment your loops with:

- the **loop variant**: a quantity that provably strictly decreases at every iteration according to some well-founded order towards some constant lower bound, thereby establishing finite execution of the loop, and
- the **invariant**: a statement that is provably true at the start of every iteration [**?**, pp. 18–20].

Data Structures

Every new data structure you implement *must* be commented with:

- the **representation convention**: an explanation of how to interpret the contents
- the **representation invariant**: a statement that is true before and after every modification.

Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2019
Report for Assignment n by Team t

Clara CLÄVER Whiz KIDD

14th November 2023

This document shows the ingredients of a good assignment report for this course. The LaTeX source code of this document illustrates almost everything you need to know about LaTeX in order to typeset a professional-looking assignment report (for this course). Use it as a starting point for imitation and delete everything irrelevant. The usage of LaTeX is *optional*, but highly recommended, for reasons that will soon become clear to those who have never used it before; any learning time is *outside* the budget of this course, but will hugely pay off, if not in this course then in the next course(s) you take and when writing a thesis or other scientific report.

Part 1
Insertion Sort

Insertion-Sort[1] "is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure **??**. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table." (Quoted from page 17 of CLRS3 [**?**].)

In the sequel of part **??**, assume that the problem tasks in the assignment were as follows (actual assignment statements in this course may have other tasks):

A. Implement Insertion-Sort as a Python function *insertion_sort*(A), where the elements to be sorted are provided in an integer array A indexed from 0 to $n-1$. The sorting is to be done *in place*, returning A in non-decreasing order, that is: $A[0] \leq A[1] \leq \cdots \leq A[n-1]$. For brevity, you can refer to a sequence in non-decreasing order as a *sorted sequence*.

B. Compute the best, average, and worst-case time complexities of Insertion-Sort.

A Specification and Program

A specification of sorting and our Python implementation of Insertion-Sort are given in Listing **??**, where n is referred to as len(A).

[1] Your report need not contain an explanation, like in this paragraph, of the problem to be solved: you can start with the task answers, assuming the reader has read the problem statement in the assignment.

Figure 1: Sorting a hand of cards using insertion sort. (© nobody, 2010)

B Complexity Analysis

The program in Listing **??** has two nested loops, so we analyse it starting from the inner loop, in lines **??** to **??**, whose purpose is to insert $A[j]$ into the *sorted* sequence $A[0\,..\,j-1]$, assuming $j > 0$, yielding the sorted sequence $A[0\,..\,j]$. Let $T_{\text{ins}}(j)$ denote the running time of this inner loop:

$$T_{\text{ins}}(j) = \begin{cases} \Theta(1) & \text{if } A[j-1] \leq A[j] & \text{(best case)} \\ \Theta(j) & \text{if } A\left[\frac{j-1}{2}\right] < A[j] \leq A\left[\frac{j+1}{2}\right] \quad (\text{if } j > 1) & \text{(average case)} \\ \Theta(j) & \text{if } A[j] < A[0] & \text{(worst case)} \end{cases}$$

assuming that every comparison takes constant time and every assignment takes constant time.

We can now analyse the outer loop, and hence the whole algorithm. Let n denote $\text{len}(A)$ and let $T(n)$ denote the running time of insertion_sort(A). We get the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < 2 \\ T(n-1) + T_{\text{ins}}(n) & \text{if } n \geq 2 \end{cases}$$

Using recurrence (**??**), we get the following time complexity results:

- $T(n) = \Theta(n)$ in the *best case*, where the array is already non-decreasingly ordered before the sorting, so that $T_{\text{ins}}(n) = \Theta(1)$ at *every* iteration of the outer loop, because $A[j]$ is always kept by the inner loop *behind* the sorted sequence $A[0\,..\,j-1]$. This result follows from Theorem **??** below, for the constants $a = 1$ and $b = 2$.

- $T(n) = \Theta(n^2)$ in the *average case*, defined here as follows: *on average* over the iterations of the outer loop, the inner loop inserts $A[j]$ into the *middle* of the sorted sequence $A[0\,..\,j-1]$, so that $T_{\text{ins}}(n) = \Theta(n)$ on average at *every* iteration of the outer loop. This can be proven by induction: ⟨insert your proof here⟩.

- $T(n) = \Theta(n^2)$ in the *worst case*, where the array is non-*increasingly* ordered before the execution of the algorithm, so that $T_{\text{ins}}(n) = \Theta(n)$ at *every* iteration of the outer loop, because $A[j]$ is always inserted by the inner loop at the *beginning* of the sorted sequence $A[0\,..\,j-1]$. This result has the same proof as in the average case above.

In conclusion, Insertion-Sort takes $\mathcal{O}\left(n^2\right)$ time for an array of n elements.

```python
def insertion_sort(A: List[int]) -> None:
    """
    Pre: (none)
    Post: A is a non-decreasingly ordered permutation of its original elements
    Ex: A = [5, 7, 3, 12, 1, 7, 2, 8, 13]
         insertion_sort(A)
         # A is now [1, 2, 3, 5, 7, 7, 8, 12, 13]
    """
    for j in range(1, len(A)):
        # Invariant: A[0..j-1] is a sorted permutation of its original elements
        # Variant: len(A) - j
        key = A[j]
        i = j - 1
        while i >= 0 and A[i] > key:
            # Invariant: A[i+2..j] has the original elements of A[i+1..j-1]
            # Variant: i
            A[i+1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Listing 1: Python implementation of the INSERTION-SORT algorithm on page 18 of CLRS3 [**?**]. ☞ Compare for example line **??** with line 5 of that algorithm: arrays are indexed from 1 in CLRS3 but from 0 in Python. Note also that a range $\ell \,..\, u$, as used in the mathematical notation of the comments, is denoted by the range $\ell : u+1$ in Python; you can use the Python notation in comments and the running text, as long as you comply with the Python semantics.

Theorem 1. *The following recurrence, for some* constants *a and b:*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < b \\ a \cdot T(n-1) + \Theta(1) & \text{if } n \geq b \end{cases}$$

has $\Theta(n)$ as closed form for $a = 1$, and $\Theta(a^n)$ as closed form for $a > 1$.

Proof. By induction (left as an exercise to the reader in the AD1 course). □

Part 2
Weighted Interval Scheduling

> "Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed *activities* that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a *start time* s_i and a finish time f_i , where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the *activity-selection problem*, we wish to select a maximum-size subset of mutually compatible activities."

(Quoted from page 415 of CLRS3 [**?**].)

The weighted interval scheduling problem is an extension to the activity-selection problem where activity a_i has the weight w_i and where an optimal solution is to be found. An optimal solution to the weighted interval problem is a subset, $O \subseteq \{a_1, a_2, \ldots, a_n\}$, where the total weights over the selected activities, $\sum \{w_i \mid a_i \in O\}$, is maximised.

Additionally, the activities are sorted in non-decreasing order of finish times: $f_1 \leq f_2 \leq \cdots \leq f_n$ and the help function $p(i)$ gives the largest index $j < i$ such that activity i and j are compatible:

$$p(i) = \begin{cases} \arg\max_{j \in 1..i-1}(s_i \geq f_j) & \text{if } \exists j \in 1..i-1 : s_i \geq f_j \\ 0 & \text{otherwise} \end{cases}$$

In the sequel of part **??**, assume that the problem tasks in the assignment were as follows (actual assignment statements in this course may have other tasks):

A. Give a recursive equation for a parameterised quantity after stating its meaning in terms of all its parameters. Use the equation to justify that the problem has the optimal substructure property and overlapping subproblems, so that dynamic programming is applicable to it.

B. Motivate your choice between bottom-up iteration (for which you must argue for the chosen nesting and iteration order of the loops) and top-down recursion for the Weighted Interval Scheduling Problem.

A Recursive equation

The weighted interval scheduling problem can be represented by the following recursive function:

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\ \{\ OPT(i-1), w_i + OPT(p(i))\ \} & \text{if } i > 0 \end{cases}$$

Given an instance of the Weighted Interval Scheduling problem with an optimal solution O, $O \subseteq J$. The last activity, a_n, either belongs to O, $a_n \in O$, or it does not, $a_n \notin O$. If a_n belongs to O, then all intervals not compatible with a_n, $\{a_i \mid i \in p(n) + 1\,..\,n-1\}$, do not belong to O.

Additionally, if a_n belongs to O, then O must include an optimal solution, $O'_{p(n)}$, to the subproblem $\{a_i \mid i \in 1\,..\,p(n)\}$. If $O'_{p(n)}$ is not optimal, then $O'_{p(n)}$ could be modified into a solution that is optimal and where all activities are intrinsically compatible with a_n.

If a_n does not belong in O, then O is an optimal solution to the subproblem $\{a_i \mid i \in 1\,..\,n-1\}$. The reasoning is analogous: assume that $a_n \notin O$; so if O is not an optimal solution to the subproblem of activities $\{a_i \mid a_i \in 1\,..\,n-1\}$, then O could be modified into a solution that is.

Deciding if a_n is to belong in O thus require an optimal solution to the subproblems $\{a_i \mid i \in 1\,..\,j\}$, with j taking the value between 1 and n, $j \in 1\,..\,n$, proving that Weighted Interval scheduling has optimal substructure.

Given the index i of an activity a_i, the recursive equation **??** returns the weight 0 for index 0, denoting an empty set of activities, and otherwise the optimal solution to the two subproblems: when activity a_i does not belong to the optimal subproblem for the activities $\{a_k \mid k \in 1\,..\,i-1\}$, or the optimal solution to the subproblem with activities $\{a_k \mid k \in 1\,..\,p(i)\}$ plus the additional weight w_i.

There are some problem instances that have overlapping subproblems. For example; given the activities $\{a_1, a_2, \ldots, a_n\}$, $p(i) = k$, and $k \geq 1$, then the subproblem $\{a_1, a_2, \ldots, a_k\}$ will be calculated at least two times, once when finding the optimal solution for the problem $OPT(n)$ and once for the subproblem $OPT(k+1)$.

B Bottom-Up Iteration and Top-Down Recursion

Given the reasoning and the recursive function in task **??**, the optimal solution to each subproblem S_i, with $1 \leq i \leq n$ and $S_i = \{a_1, \ldots, a_i\}$, will be found, giving bottom-up iteration and top-down recursion the same time complexity. We have chosen the bottom-up iterative approach. Our program uses a single loop where all activities are iterated over in increasing order of their indices, where activity a_i has index i.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

More LaTeX and Technical Writing Advice

Unnumbered itemisation (only to be used when the order of the items does *not* matter):[2]

- Unnumbered displayed formula:
$$E = m \cdot c^2$$
- Numbered displayed formula, which is cross-referenced somewhere:
$$E = m \cdot c^2$$
- Formula — the same as formula (**??**) — spanning more than one line:
$$\begin{aligned} E \\ &= m \cdot c^2 \end{aligned}$$

Numbered itemisation (only to be used when the order of the items *does* matter):

1. First do this.
2. Then do that.
3. If we are not finished, then go back to Step **??**, else stop.

Tables and elementary mathematics are typeset as given in Table **??**; see `ftp://ftp.ams.org/pub/tex/doc/amsmath/short-math-guide.pdf` for many more details.

Use `\mathit{...}` in mathematical mode for each multiple-letter identifier in order to avoid typesetting the identifier like the product of single-letter ones. For example, note the typographic difference between the identifier WL, obtained through `$\mathit{WL}$`, and the product WL, where there is a small space between the W and the L, obtained through `$WL$`.

Do *not* use programming-language-style lower-ASCII notation (such as ! for negation, && for conjunction, || for disjunction, and the equality sign = for assignment) in algorithms or formulas (but rather use $\neg$ or **not**, $\wedge$ or & or **and**, $\vee$ or **or**, and $\leftarrow$ or $:=$, respectively), as this testifies to a very strong confusion of concepts.

Figures can be imported with `\includegraphics` or drawn inside the LaTeX source code using the highly declarative notation of the `tikz` package: see Figure **??** for sample drawings. It is perfectly acceptable in this course to include scans or photos of drawings that were carefully done by hand.

If you are not sure whether you will stick to your current choice of notation or terminology, then introduce a new (possibly parametric) command. For example, upon

```
\newcommand{\Cardinality}[1]{\left\lvert#1\right\rvert}
```

the formula `$\Cardinality{S}$` typesets the cardinality of set S as $|S|$ with autosized vertical bars and proper spacing, but upon changing the definition of that parametric command to

```
\newcommand{\Cardinality}[1]{\# #1}
```

[2] Use footnotes very sparingly, and note that footnote pointers are *never* preceded by a space and always glued immediately *behind* the punctuation, if there is any.

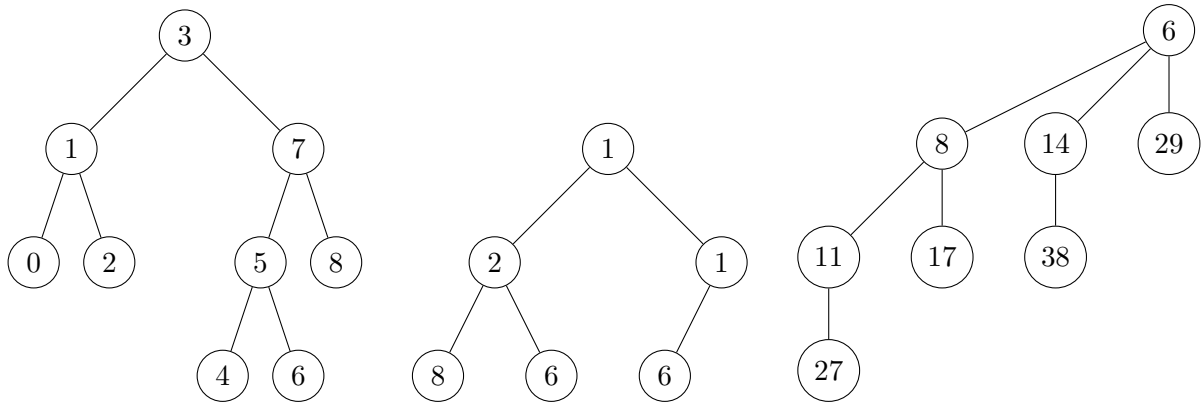

Figure 2: A binary search tree (on the left), a binary min-heap (in the middle), and a binomial tree of rank 3 (on the right).

and recompiling, the formula `$\Cardinality{S}$` typesets the cardinality of set S as $\#S$. You can thus obtain an arbitrary number of changes in the document with a *constant*-time change in its source code, rather than having to perform a *linear*-time find-and-replace operation within the source code, which is painstaking and error-prone. The source code of this document has some useful predefined commands about mathematics and algorithms.

Use commands on positioning (such as `\hspace`, `\vspace`, and `\noindent`) and appearance (such as `\small` for reducing the font size, and `\textit` for italics) very sparingly, and ideally only in (parametric) commands, as the very idea of mark-up languages such as LaTeX is to let the class designer (usually a trained professional typesetter) decide on where things appear and how they look. For example, `\emph` (for emphasis) compiles (outside italicised environments, such as `theorem`) into *italics* under the `article` class used for this document, but it may compile into **boldface** under some other class.

If you do not (need to) worry about *how* things look,
then you can fully focus on *what* you are trying to express!

Note that *no* absolute numbers are used in the LaTeX source code for any of the references inside this document. For ease of maintenance, `\label` is used for giving a label to something that is automatically numbered (such as an algorithm, equation, figure, footnote, item, line, part, section, subsection, or table), and `\ref` is used for referring to a label. An item in the bibliography file is referred to by `\cite` instead. Upon changing the text, it suffices to recompile, once or twice, and possibly to run BibTeX again, in order to update all references consistently.

Always write `Table~\ref{tab:maths}` instead of `Table \ref{tab:maths}`, by using the non-breaking space (which is typeset as the tilde ~) instead of the normal space, because this avoids that a cross-reference is spread across a line break, as for example in "Table **??**", which is considered poor typesetting.

The rules of English for how many spaces to use before and after various symbols are given in Table **??**. Beware that they may be very different from the rules in your native language.

		number of spaces after	
		0	1
number of spaces before	0	/ -	, : ; . ! ?)] } ’ ” %
	1	([{ ‘ “	– (n-dash) — (m-dash)

Table 1: Spacing rules of English

Topic	LaTeX code	Appearance
Greek letter	`$\Theta,\Omega,\epsilon$`	Θ, Ω, ϵ
multiplication	`$m \cdot n$`	$m \cdot n$
division	`$\frac{m}{n}, m \div n$`	$\frac{m}{n}, m \div n$
rounding down	`$\left\lfloor n \right\rfloor$`	$\left\lfloor n \right\rfloor$
rounding up	`$\left\lceil n \right\rceil$`	$\left\lceil n \right\rceil$
binary modulus	`$m \bmod n$`	$m \bmod n$
unary modulus	`$m \equiv n \mod \ell$`	$m \equiv n \mod \ell$
root	`$\sqrt{n},\sqrt[3]{n}$`	$\sqrt{n}, \sqrt[3]{n}$
exponentiation, superscript	`$n^{i}$`	n^{i}
subscript	`$n_{i}$`	n_{i}
overline	`$\overline{n}$`	$\overline{n}$
base 2 logarithm	`$\lg n$`	$\lg n$
base b logarithm	`$\log_b n$`	$\log_b n$
binomial	`$\binom{n}{k}$`	$\binom{n}{k}$
sum	`\[\sum_{i=1}^n i\]`	$\sum_{i=1}^{n} i$
numeric comparison	`$\leq,<,=,\neq,>,\geq$`	$\leq, <, =, \neq, >, \geq$
non-numeric comparison	`$\prec,\nprec,\preceq,\succeq$`	$\prec, \nprec, \preceq, \succeq$
extremum	`$\min,\max,+\infty,\bot,\top$`	$\min, \max, +\infty, \bot, \top$
function	`$f\colon A\to B,\circ,\mapsto$`	$f\colon A \to B, \circ, \mapsto$
sequence, tuple	`$\langle a,b,c \rangle$`	$\langle a, b, c \rangle$
set	`$\{a,b,c\},\emptyset,\mathbb{N}$`	$\{a, b, c\}, \emptyset, \mathbb{N}$
set membership	`$\in,\not\in$`	$\in, \notin$
set comprehension	`$\{i \mid 1 \leq i \leq n\}$`	$\{i \mid 1 \leq i \leq n\}$
set operation	`$\cup,\cap,\setminus,\times$`	$\cup, \cap, \setminus, \times$
set comparison	`$\subset,\subseteq,\not\supset$`	$\subset, \subseteq, \not\supset$
logic quantifier	`$\forall,\exists,\nexists$`	$\forall, \exists, \nexists$
logic connective	`$\land,\lor,\neg,\Rightarrow$`	$\land, \lor, \neg, \Rightarrow$
logic	`$\models,\equiv,\vdash$`	$\models, \equiv, \vdash$
miscellaneous	`$\&,\#,\approx,\sim,\ell$`	$\&, \#, \approx, \sim, \ell$
dots	`$\ldots,\cdots,\vdots,\ddots$`	$\ldots, \cdots, \vdots, \ddots$
dots (context-sensitive)	`$1,\dots,n; 1+\dots+n$`	$1, \dots, n; 1 + \dots + n$
parentheses (autosizing)	`$\left(m^{n^k}\right),(m^{n^k})$`	$\left(m^{n^k}\right), (m^{n^k})$
identifier of > 1 character	`$\mathit{identifier}$`	$\mathit{identifier}$
hyphen, n-dash, m-dash, minus	`-, --, ---, $-$`	-, –, —, −

Table 2: The typesetting of elementary mathematics. Note very carefully when italics are used by LaTeX and when not, as well as all the horizontal and vertical spacing performed by LaTeX.