COMP3711: Design & Analysis of Algorithms

2021 Fall Semester, HKUST

By Ljm

This is the note of course **COMP3711:** *Design & Analysis of Algorithms* offered in 2021 Fall semester in HKUST by Professor GOLIN, Mordecai Jay and Professor CHENG, Siu Wing. The content is almost totally written by Ljm, with some code snippets and images chosen from course slides as well as the textbook *Introduction to Algorithms, 3ed*. You can find more details on Canvas website for this course, `https://canvas.ust.hk/courses/38226`, hopefully no login is required.

I have written this note in a very short space of time and as a result it may contain many errors and inaccuracies, readers are welcome to email me at `enor2017@163.com` or create an issue on my GitHub repo `https://github.com/enor2017/CourseNotes/issues`.

Hope you enjoy this book!

Contents

I Foundations

0. Introduction to Algorithms

0.1 What is an Algorithm?

Definition 0.1.1 An **algorithm** is a recipe for doing something. An **algorithm** is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions.

Let's look at the algorithm that adding two numbers, which you use every day.

Input: Two numbers x and y, each consisting of n digits: $x = \overline{x_n x_{n-1} \cdots x_1}, y = \overline{y_n y_{n-1} \cdots y_1}$. They maybe very long.

Output: A number $z = \overline{z_{n+1} z_n \cdots z_1}$ such that $z = x + y$.

The pseudocode is given below:

Algorithm 1: Add-Two-Numbers(x, y)

```
1  c ← 0        // c is carry-in
2  for i ← 1 to n do
3  |   z_i ← x_i + y_i + c
4  |   if z_i ≥ 10 then
5  |   |   c ← 1, z_i ← z_i − 10
6  |   else
7  |   |   c ← 0
8  |   end
9  end
10 z_{n+1} ← c
```

0.2 Insertion Sort

Problem: Given an array $A[1 \cdots n]$ of elements, sort the array in ascending order.

Well, to design an algorithm, what I typically do is to think how we will solve this problem by hand. Consider when you play poker, you already have a hand of sorted cards, and when you get a new card, you *choose a proper position to insert it*. For example, you currently have cards:

$$[1, 3, 4, 5, 9]$$

and then you get a new card 7, you scan over the cards, and find out that you should insert 7 between 5 and 9. This is basically how we do in insertion sort algorithm.

Let's do a more completed demo. At the beginning, you have an array

$$[5, 1, 8, 3]$$

At step 1, you get 5, since you have no number before, 5 itself is already sorted, so now in your hand, you have

$$[5]$$

At step 2, you get 1, and you find out 1 should be inserted before 5, since $1 < 5$. Now in your hand, you have

$$[1, 5]$$

At step 3, you get 8, and you find out 8 should be inserted after 5, so now in your hand,

$$[1, 5, 8]$$

At step 4, you get 3, it should be inserted between 1 and 5, so now in your hand,

$$[1, 3, 5, 8]$$

This is not difficult right? However, to translate it into a formal algorithm, we need to figure out two things:

- How should we find the position to insert a number, and
- How should we actually "insert" the number into the array?

For the first point, when we get a new number x, we can scan the numbers we already had, from right to left, until we meet a number y such that $y \leq x$, then we know we should insert x after y.

For the second point, since this is an array, the only thing we can do is to "move all items one position afterwards", and then put the new number into the vacant position. For example, insert 3 into $[1, 5, 8]$, we first move 5 and 8 backwards, say

$$[1, \_, 5, 8]$$

then, put 3 inside,

$$[1, 3, 5, 8]$$

Alternatively, we can continuously swap from right to left, until $y \leq x$ is met. Notice that this method is preferred, since we combine the process of "finding insertion position" and "insert number into array" into one! As an example, insert 3 into $[1, 5, 8]$, we first append 3 at last,

$$[1, 5, 8, 3]$$

since $8 > 3$, we swap 8 with 3,

$$[1, 5, 3, 8]$$

, since $5 > 3$, we swap 5 with 3,

$$[1, 3, 5, 8]$$

, since $1 < 3$, now 3 is at the correct position, no further swap is required.

Up to now, you should have known the basic idea as well as some different implementations of Insertion Sort algorithm. To describe it using the last method with pseudocode,

Algorithm 2: Insertion-Sort(A)

```
   // don't need to sort first item, so we proceed with second item, here i
      points to the new item we get
1  for i ← 2 to n do
2  |  j ← i − 1      // j points to right most item
   |  // As long as A[j] > A[j + 1], we continuously swap j and j + 1
3  |  while j > 1 and A[j] > A[j + 1] do
4  |  |  swap A[j] and A[j + 1]
5  |  |  j ← j − 1      // continue to move left, j decreases
6  |  end
7  end
```

After designing an algorithm, we need to prove its correctness, which, in this example, is to prove this algorithm can actually sort items in ascending order. The correctness of this algorithm is hopefully, obvious, and you may find a very brief proof on lecture slide, you can also prove it by induction.

Apart from correctness, we also need to examine its performance. Usually, we care about its **time complexity** and **space complexity**, you can think of them as, respectively, how long

will the algorithm take for a certain amount of input, and how much space/memory will the algorithm occupy during execution. We will formally introduce them in next topic.

For now, (and actually most time in this course), we only examine the running time, i.e., **time complexity** of Insertion Sort. Hopefully, it is not surprising that *line 3* above is the dominant part of running time, since it resides in the most inner part of the two loops. Thus, we can simply calculate how many times will line 3 be executed, and use it to serve as the running time of this algorithm.

Let's have a look at line 3, the "while" loop will repeat as long as $j \geq 1$ and $A[j] > A[j+1]$, i.e., as long as the current number is smaller than the left number, we will swap them, and continue to check next one. So the extreme case is if the current number, $A[j]$, is the *smallest* among all previous numbers, $A[1 \cdots j-1]$, then we will perform $j-1$ swaps. Therefore, this line will run at most

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

times. This immediately follows that **in the worst case, the running time of Insertion Sort is** $\frac{n(n-1)}{2}$**.**

However, that's the worst case, how about other cases? Let's now assume the opposite to what we assumed just now, that is, if the current number, $A[j]$, is the *largest* among all previous numbers, $A[1 \cdots j-1]$, then we will not perform any swaps, so line 3 only run $(n-1)$ times.(every time we do a checking and quit the while loop immediately)

From above analysis, we can observe that in different cases, i.e., when the data vary, the running time of an algorithm also varies. We will come back to this point in next topic. One thing worths mentioning is that in our worst case above, we assume the current item is the smallest one, so the input data is "*reversely sorted*"; will in our best case above, we assume the current one is largest, so the input data is *already sorted*.

0.3 About pseudocode

In this course, we use pseudocode to explicitly describe algorithms. There are lots of reasons for this. On one hand, real codes, such as Java, C++ programs, are too hard to read and implement. For example, "sort n intervals according to their start time" is not easy to implement in most programming languages, but in pseudocode, we can just write like this without any ambiguity. On the other side, if we use natural languages, they are, usually, not descriptive enough, or they may be quite ambiguous sometimes.

Pseudocode codes have different styles, and in this course, we normally follow the below rules:

- **Use standard keywords**: if/then/else, for, while, return, repeat/until
- **Use standard notations**:
  - Use "$variable \leftarrow value$" to assign value to a variable,
  - Use "$variable = value$" as an "assertion", which is a true-of-false statement to check if they are equal
  - Use "$Array[index]$" to represent arrays,
  - Use "$function(arguments)$" to represents functions, $\cdots$
- **Make clear indentations**: If you know Python, you know how to do it
- **Use mathematical notations**:
  - Use "$i \leftarrow i + 1$" instead of $i = i + 1$, or $i++$
  - Use "$x \cdot y, x \bmod y$" instead of $x * y, x\%y$
  - Use "$\sqrt{x}, a^b$" instead of sqrt(x), power(a, b)
  - Use "$\lfloor n/2 \rfloor$" instead of $n/2$: we don't have implicit truncation
  - Use "remove first item from list" instead of List.removeFirst()
- **Hide unnecessary data structures/sub-routines, make them becomes "black-box"**:
  - Directly say "use Insertion Sort to sort array $A[1 \cdots n]$", instead of re-write whole procedure of Insertion Sort.
  - Directly say "create a heap", instead of re-describe how to construct and implement a heap
  - Directly say "$x =$ the maximum element in A" or $M = \max_{1 \le i \le n} A[i]$ instead of writing a loop to find. (But notice this takes $O(n)$ time, not $O(1)$)

In order to write pseudocode for this course, I found it handy to type it in LaTeX, as an example, the code below

```
\begin{algorithm*}[htbp]
    \caption{Example-Pseudocode($A[1\cdots n]$)}

    \tcp{This is a comment.}

    $result \leftarrow 0$

    \For{$i\leftarrow 1$ to $n$}{
        \eIf{$i=1$}{
            $result \leftarrow result + 1$
        }{
            $result \leftarrow result + i$
```

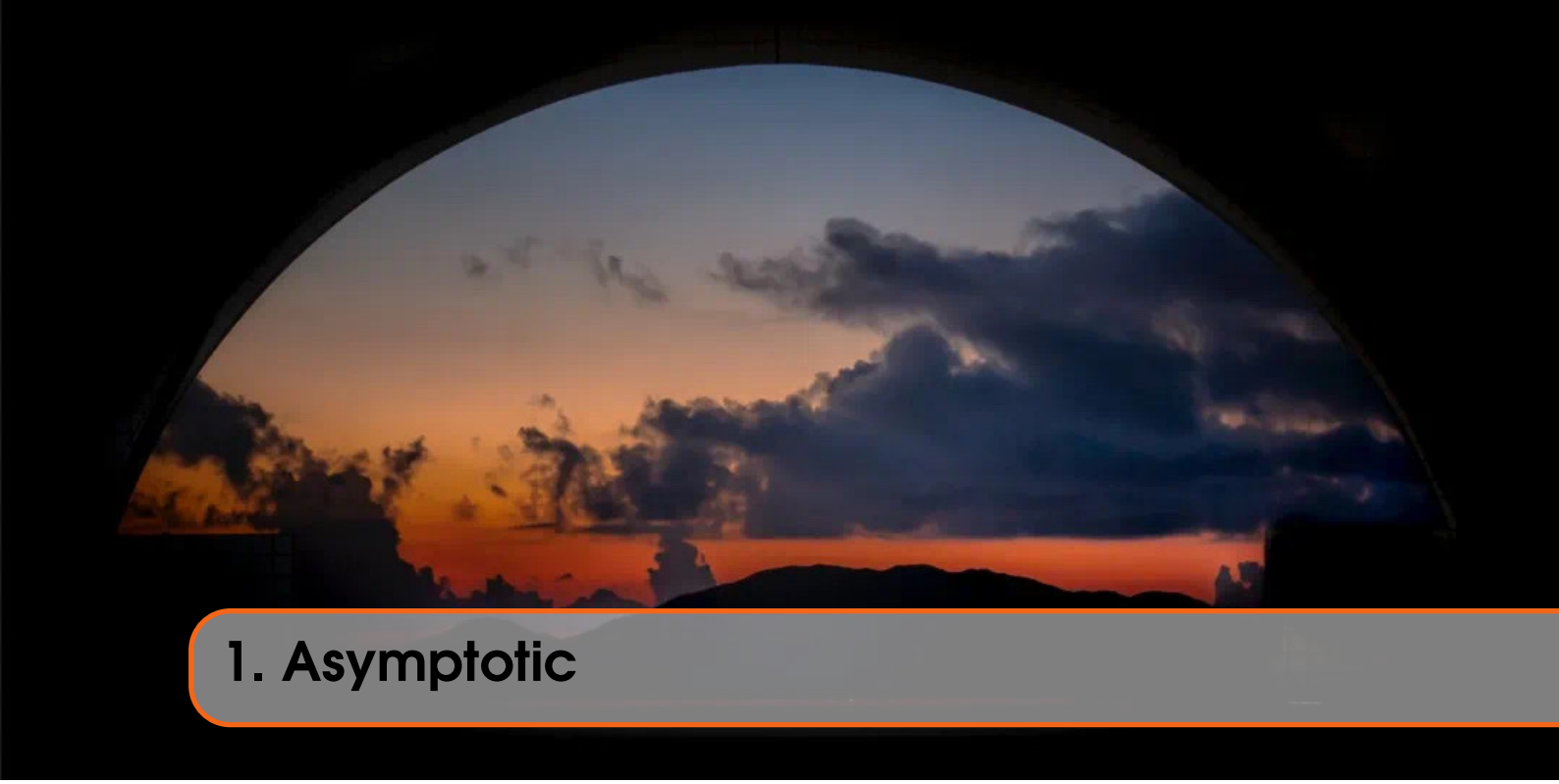
```
13          }
14          \If{$result > 1000$}{
15              break
16          }
17      }
18      return $result$
19  \end{algorithm*}
```

will be rendered as:

Algorithm 3: Example-Pseudocode($A[1 \cdots n]$)

```
   // This is a comment.
 1 result ← 0
 2 for i ← 1 to n do
 3 |   if i = 1 then
 4 |   |   result ← result + 1
 5 |   else
 6 |   |   result ← result + i
 7 |   end
 8 |   if result > 1000 then
 9 |   |   break
10 |   end
11 end
12 return result
```

I believe the syntax is relatively easy to understand.



1. Asymptotic

1.1 Algorithm

An **algorithm** is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. To evaluate an algorithm, we measure:

- **Memory (Space Complexity)**: all space used except for holding inputs
- **Running time (Time Complexity)**

In this course, we measure algorithms *analytically*, i.e., depends only on the algorithms, without considering actual implementations, hardwares, etc.

However, it is difficult and rarely that we can say "one algo is better than the other", since that usually depends on input size, input data(even for same size), etc.

1.2 Time Complexity

Usually, we measure running time(time complexity) as the num of machine instructions, such as addition, multiplication, swap(as used in sorting analysis), etc. We describe running time as a function of input size: $T(n)$.

There are three commonly-used asymptotic notations:

Definition 1.2.1 — Upper bounds. $T(n) = O(f(n))$, if $\exists c > 0, n_0 \geq 0$ such that $\forall n \geq n_0, T(n) \leq c \cdot f(n)$.

Definition 1.2.2 — Lower bounds. $T(n) = \Omega(f(n))$, if $f(n) = O(T(n))$.

Definition 1.2.3 — Tight bounds. $T(n) = \Theta(f(n))$, if both $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Here are some notes for above notations: First, more accurate expression should be $T(n) \in O(f(n))$, but we often use $=$ for simplicity, which means "is", not "equal". Second, these notations is not properly definable using limits. One may think that $f(n) = O(g(n))$ is equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, but a counterexample can easily be found like $f(n) = (2 + (-1)^n)g(n)$, in which case the limit does not exist.

I will omit examples here, but I'd like to list some interesting facts.

1. 2^{10n} is not $O(2^n)$, since it is $(2^n)^{10}$.
2. $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$.
3. $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$, which is called *Harmonic Series*.
4. $\log(n!) = \Theta(n \log n)$.

For a certain algorithm, different inputs can cause different performances, even with same input size. For insertion sort, input an already sorted list requires no additional swaps, which gives $\Theta(n)$, and this is called **best case**; input an inversely sorted list gives $T(n) = \sum_{i=2}^{n}(i-1) = \Theta(n^2)$, this is **worst case**; if we average over all possible inputs for a certain size n, assuming same probability distribution on these inputs, then the result running time is called **average case**.

Generally, average case analysis is rather complicated. In insertion sort, we assume each of the $n!$ permutations is distributed equally likely. With some probability knowledge we will know it's $\Theta(n^2)$. I will give brief proof in last page of this note if you are interested.

Let's have a summary of three kinds of analysis: (1) best case is ideal so that it is useless; (2) average case is sometimes used but requires complicated analysis; (3) **worst case** is commonly used, since it gives running time guarantee **independent of actual input**. In this course, **Worst-case analysis is the default**, but it is not perfect: some algorithms with bad worst-case running time actually work very well in practice, since worst case input rarely occurs.

When we say an algorithm's worst case running time is $O(f(n))$, we mean **on all inputs of size** n, the algorithm's running time is $O(f(n))$, but there is no need to really *find* the worst input to prove.

When we say an algorithm's worst case running time is $\Omega(f(n))$, we mean **there exists at least one input of size** n, the algorithm's running time is $\geq c \cdot f(n)$. We mainly use this to prove the big-Oh analysis is tight.

To understand above two paragraphs, again consider insertion sort: it runs in $\leq \frac{n(n-1)}{2}$ time for all inputs of size n, so it is $O(n^2)$, it **requires** $\frac{n(n-1)}{2}$ time if items are reversed, so it is $\Omega(n^2)$. To combine, it runs in $\Theta(n^2)$ time.

Brief proof for average case time complexity of insertion sort:(totally optional for this course)

Firstly, one can show that the number of "swaps" is equals to the number of **inversions**.(proof by induction in lecture slide divide & conquer)

So now we know the running time for a certain input will be $\Theta(n + I)$, where I is the number of inversions of the original array.

Here, we define X_{ij} to be 1 if $a[i]$ and $a[j]$ form an inversion and 0 otherwise. So an given input of size n will have $\frac{n(n-1)}{2}$ different X_{ij}s.

Now, we can express I as: $I = \sum X_{ij}$. But remember we are interested in the **expected number of inversions** in the array, since we're looking for average running time of all inputs. This is also simple by linearity of expectation:

$$\mathbb{E}(I) = \mathbb{E}\left(\sum X_{ij}\right) = \sum \mathbb{E}(X_{ij})$$

That's a good one, $\mathbb{E}(X_{ij})$ is the expected value of X_{ij}, of course it is $1 \cdot P(X_{ij} = 1) = 0.5$, since we have assumed $n!$ permutations are equally likely.

Thus, $\mathbb{E}(I) = \sum \frac{1}{2}$, and there are $\frac{n(n-1)}{2}$ terms, which gives $\mathbb{E}(I) = \frac{n(n-1)}{4} = \Theta(n^2)$.

To sum up, on expectation the runtime will be $\Theta(n^2 + n) = \Theta(n^2)$, This explains why the average-case behavior of insertion sort is $\Theta(n^2)$.

2. Divide and Conquer

2.1 Intro: Binary Search

The main idea of Divide & Conquer is to solve a problem(such as of size n) by breaking it into one or more smaller(size less than n) problems. We use binary search example to illustrate that.

Problem: given an **sorted** array of length n, how to find the position of element x; if x does not exist in the array, output nil.

Since the array is already sorted, it has a good property that: **for each item a_i, those who are larger than a_i must be on its right side, while smaller than a_i must be on its left side.** Hence we come up with an idea that we check the middle item mid first, then we will be able to know which direction to go: left or right, depending on the comparison of mid and x(the item we're looking for). If we go left, then the right half will be directly abandoned. Then we continue this process, check middle item each time, and abandon half items each time.

Algorithm 4: BinarySearch($a[]$, $left$, $right$, x)

```
Data: a[]: the array given, x: the item to find
if left = right then
    if a[left] = x then
        return left
    else
        return nil
    end
else
    mid = ⌊(left + right)/2⌋
    if x ≤ a[mid] then
        BinarySearch(a[], left, mid, x)
    else
        BinarySearch(a[], mid + 1, right, x)
    end
end
```

First call: BinarySearch($a[]$, 1, n, x).

This algorithm is quite efficient, since each time we eliminate half of the array, with one additional comparison, until there is only one item left, when we will end the process.

Then let's analyse its time complexity. Let $T(n)$ be the number of comparisons needed for n elements, then we will have

$$T(n) = T(n/2) + 1,\ T(1) = 1$$

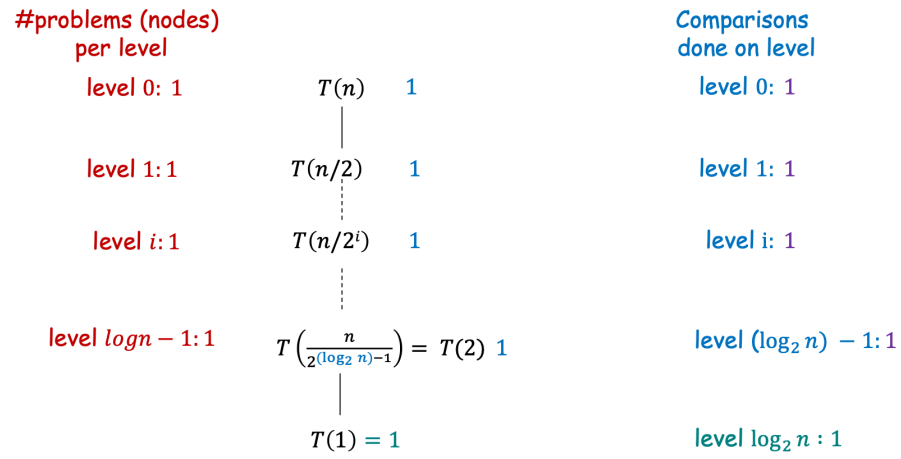
.

Solve this **recurrence**:

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= [T(n/4) + 1] + 1 \\ &= T(n/4) + 2 \\ &= \cdots \\ &= T(n/2^i) + i \end{aligned}$$

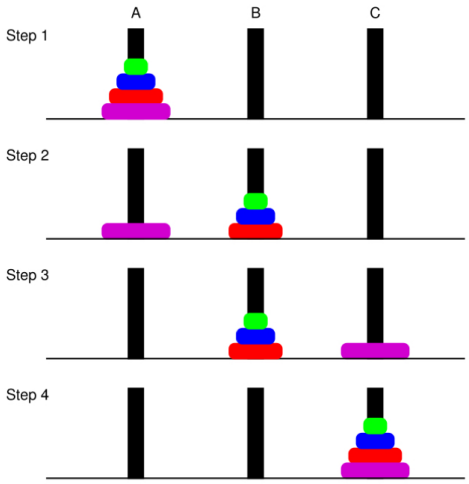
This process ends when reaching $T(1)$, i.e., $i = \log_2 n$, thus, $T(n) = T(1) + \log_2 n = \log_2 n + 1$.

We can also visualize this recurrence with recursion tree: (image from lecture note)



In each recursion step(level), we use 1 comparison(compare mid and x), then call recursion on a half of the original array. From the image above, we can easily notice there are total $1 + 1 + \cdots + 1 = 1 + \log_2 n$ comparisons.

2.2 Example: Towers of Hanoi



In this example, we want to design an algorithm to move all n discs from peg A(start) to peg C(end), with the constraints: (1) move one disc at a time, and (2) cannot put larger disc on a smaller one. We are given another peg B(helper) where we can temporary storage our discs.

We still use the idea of **Divide & Conquer**, consider how we can turn a problem of n discs into a problem of $n-1$? One possible solution is that, we can call recursion on upper $n-1$ discs, i.e., move upper $n-1$ discs to peg B(helper peg), then move the remaining (the biggest) disc to peg C(end peg), and finally move the $n-1$ discs from peg B(helper) to peg C(end). The following pseudocode shows this idea.

Algorithm 5: MoveTower(n, *start*, *helper*, *end*)

```
   Input: n: num of discs
1  if n = 1 then
2  |  move the only disc from start peg to end peg
3  |  return
4  else
   |  // move first n − 1 from start peg to helper peg
   |  // so this time "helper" peg will be the old end peg
5  |  MoveTower(n − 1, start, end, helper)
6  |  move the only disc from start peg to end peg
   |  // finally move first n − 1 from helper peg to end peg
   |  // this time "helper" peg will be the old start peg
7  |  MoveTower(n − 1, helper, start, end)
8  end
```

Now we would like to analyze the time complexity of this algorithm, in other words, how many **steps** are needed. Let $T(n)$ be the num of steps for n discs, each time, we first move $n-1$ disks from *start* to *helper*, costs $T(n-1)$ steps; then we move the biggest disk to *end* peg, costs only

1 step; finally we move $n-1$ disks from *helper* to *end*, again costs $T(n-1)$ steps. To sum up:

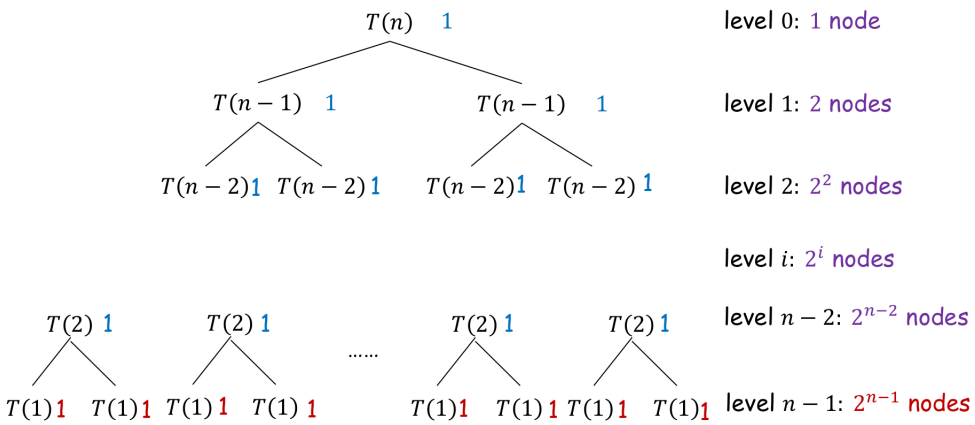
$$T(n) = 2T(n-1) + 1$$

when $n > 1$, and $T(1) = 1$.

Now we solve the recurrence by the **expansion method**:

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2[2T(n-2) + 1] + 1 \\
&= 4T(n-2) + 3 \\
&= 4[2T(n-3) + 1] + 3 \\
&= 8T(n-3) + 7 \\
&= \cdots \\
&= 2^i T(n-i) + (2^i - 1) \\
&= 2^{n-1} T(1) + (2^{n-1} - 1) \\
&= 2^n - 1
\end{aligned}$$

Or, with the recursion tree method:



There are, altogether, $1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-2} + 2^{n-1} = 2^n - 1$ nodes, and we are doing one work(step) each node, then the time complexity is again, $2^n - 1$.

2.3 Merge Sort

Now we again back to sorting, and we would like to introduce a new algorithm or sorting: Merge Sort. This is a typical example of divide & conquer, and its process is like: (1) we first divide array into two halves, (2) then we recursively sort each half, (which means we continuously divide it into halves, and then halves...) (3) finally **merge** two halves to get a whole.

The **merge** operation may confuse you most. Here it means combine two **sorted lists** into a whole sorted list. For example, given two sorted lists: $A = [2, 5, 7]$ and $B = [3, 4, 6, 10, 12]$, then after **merge** operation, we will get $result = [2, 3, 4, 5, 6, 7, 10, 12]$. Since these two lists are sorted, we can do this process in $O(n)$ time, where n is the length of result list.(how many numbers in total) The basic idea is: we compare the first item of A and B, put the smaller one, say, $A[1]$, in the first position of result list, then we move on to the next item of A, but compare it still with the **first** item of B(since the first item of B has not yet inserted into result list), and again put the smaller one into result list, then continue move on. An example may help you understand the process:

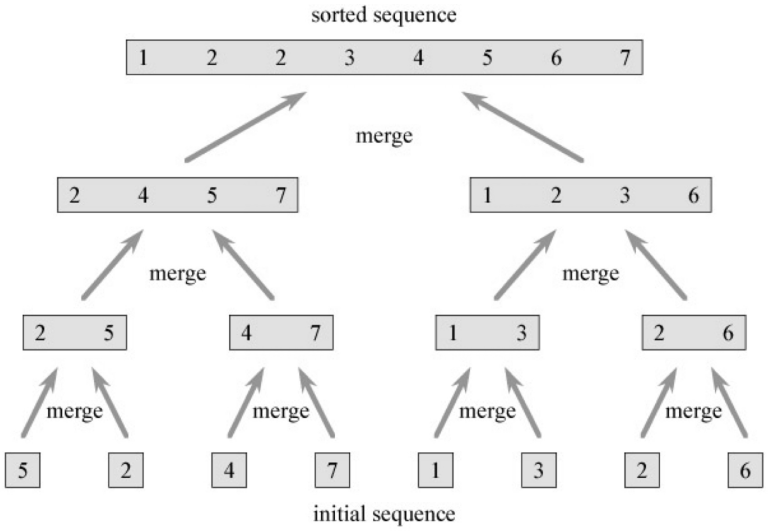
(1) Compare first items: $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $2 < 3$, so $result = [2]$;
(2) then compare 2nd in A and 1st in B, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $3 < 5$, so $result = [2, 3]$;
(3) continue the process, similarly, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $4 < 5$, so $result = [2, 3, 4]$;
(4) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $5 < 6$, so $result = [2, 3, 4, 5]$;
(5) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $6 < 7$, so $result = [2, 3, 4, 5, 6]$;
(6) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $7 < 10$, so $result = [2, 3, 4, 5, 6, 7]$;
(7) Now, all items in A have already been inserted into result list so that no items can be compared with items in B. Then we simply add remaining items in B to result list, this will, obviously, ensure a sorted result list.(you may think of why) Hence $result = [2, 3, 4, 5, 6, 7, 10, 12]$

The pseudocode below shows the process: (below, append ∞ at the end of two lists can free us from considering the situation that one list is empty, like (7) above. Though different implementation, the idea is entirely the same)

Algorithm 6: Merge(A, $left$, mid, $right$)

```
   // merge two sorted list:  A[left ··· mid] and A[mid + 1 ··· right]
 1 L ← A[left ··· mid], R ← A[mid + 1 ··· right]
 2 append ∞ at the end of L and R      // see explanation above
 3 i ← 1, j ← 1     // two pointers point at items in L and R
 4 for k ← left to right do
       // always choose the smaller one to insert, and move on
 5     if L[i] ≤ R[j] then
 6         A[k] ← L[i]
 7         i ← i + 1
 8     else
 9         A[k] ← R[j]
10         j ← j + 1
11     end
12 end
```

After learning how **Merge** works, you now, hopefully, are able to understand how Merge Sort works, with the image below:



We break down array recursively, until one element left, and then merge from bottom to up. The complete pseudocode for Merge Sort is given below:

Algorithm 7: MergeSort(A, $left$, $right$)

```
1 if left = right then
2 |  return
3 end
4 mid ← ⌊(left + right)/2⌋
  // recursively divide array into two halves
5 MergeSort(A, left, mid)
6 MergeSort(A, mid + 1, right)
  // then merge from bottom to up
7 Merge(A, left, mid, right)
```

Firstly call **MergeSort(A, 1, n)** to sort array A.

As usual, we are interested in the running time of Merge Sort algorithm. Let $T(n)$ be the running time on an array of size n, it's not hard to find $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$, when $n > 1$ and $T(1) = O(1)$.

Here we are actually able to simplify the equation. Firstly we can replace $\leq$ with $=$, since we are interested in big-O upper bound of $T(n)$; and with the same reason, we can replace $O(n)$ with n, $O(1)$ with 1; finally, we can assume n is a power of 2 for the sake of simplicity but doesn't change the result at all, as $T(n) \leq T(n') \leq T(2n) = O(T(n))$ where n' is the smallest power of 2 such that $n' \geq n$.

Now we want to solve: $T(n) = 2T(n/2) + n$ for $n > 1$, and $T(1) = 1$.

$$\begin{aligned} T(n) &= 2\left(\frac{n}{2}\right) + n \\ &= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2n \\ &= 2^2 \cdot \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3n \\ &= \cdots \\ &= 2^k \cdot T\left(\frac{n}{2^k}\right) + kn \end{aligned}$$

We know the process ends with $\dfrac{n}{2^k} = 1$ i.e. $k = \log_2 n$, thus

$$\begin{aligned} T(n) &= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + n \cdot \log_2 n \\ &= n \log_2 n + n \end{aligned}$$

In summary, merge sort runs in $O(n \log n)$ time. It is also worth pointing out that merge sort **always** runs in $O(n \log n)$ time, which means best case is the same as worst case, as you may think of it, the complexity of merge sort *does not depend on inputs*, it always break array down and then merge up.

2.4 Inversion Numbers

Given an array $A[1 \cdots n]$, we say two elements $A[i]$ and $A[j]$ are **inverted** if $i < j$ but $A[i] > A[j]$, i.e., $A[i]$ appears before $A[j]$ but is larger than $A[j]$. The number of inverted pairs is called the **inversion number** of array A. Actually this is a useful measure, it provides us with an intuitive idea about how "sorted" an array is, larger inversion number implies a more unsorted array.

What may surprise you is that inversion number has a close relation to insertion sort, and more concretely, **the number of swaps used by insertion sort is equals to inversion number.** We can prove it by induction:

Proof. Assuming the array has size n. Basic case $n = 2$ obviously holds.

Inductive step: assume correct for an array of size $n-1$, i.e., the total number of swaps performed while insertion sorting $A[1 \cdots n-1]$ is equals to the inversion number of $A[1 \cdots n-1]$.

Let $x = A[n]$. Now, the remaining work by insertion sort is that we swap x with all items $A[j]$ such that $j < n$ and $A[j] > x$, notice that the number of those items is the same of inversions in which x **participates**. Therefore, adding these new inversions gives the full inversion number of $A[1 \cdots n]$. ■

Now we will consider how to compute the inversion number of a given array with size n. One possible method is we check all (i, j) pairs of given array, this requires $\binom{n}{2} = \Theta(n^2)$ running time. Another method uses the relation we proved above, running insertion sort and count the number of swaps we perform, but this also requires $\Theta(n^2)$ time since insertion sort requires $\Theta(n^2)$. How can we improve that? Come back to topic: divide and conquer!

Similar to previous problems, we divide array into two halves, and recursively count inversions in each half, but notice: we are missing something: we still need to count inversions where a_i and a_j are in different halves! We need to return the sum of those three quantities finally.

So the main problems is that, how we count the third quantity? Consider below situation, the two halves of array are: $[1, 5, 4, 8, 10, 2]$ and $[6, 9, 12, 11, 3, 7]$, how would you do that? You may count by hand, and knowing there are $5-3, 4-3, 8-6, \cdots$ and in total 9 inversions with one item in 1st array and another in 2nd. But, it's really time consuming and totally a mess! We have no efficient algorithm to do this but to count one by one.

Fortunately, things will become much better if those two arrays are *sorted*. For example, $A = [3, 7, 10, 14, 18, 19]$ and $B = [2, 11, 16, 17, 23, 25]$. How will we do then? We can scan progressively through both lists, and for each item in B, we only need to find the smallest A item larger than it. In the lists above, for example, $A[1] = 3$ is larger than $B[1] = 2$, so all items

in A form an inversion pair with $B[1]$; then we move to $B[2] = 11$, we try to find the smallest item in A larger than 11, so we move the pointer in A, $A[2] = 7 < 11, A[3] = 10 < 11$, until $A[4] = 14 > 11$, so each item in $A[4] \cdots A[6]$ can form an inversion pair with $B[2]$. If we continue the process, we will finally get the inversion number formed between A and B, in $O(n)$ time. (Why is $O(n)$? Since we only iterate each item once during the whole process. You may find it quite similar to Merge operation in Merge Sort)

Algorithm 8: Count(A, l, mid, r)

```
  // l means left, while r means right.
  // notice here, L[1 ··· (mid − l + 1)] is corresponding to A[l ··· mid], the
     subscript changes, try not be confused later.  R also changes.
 1 L ← A[l ··· mid], R ← A[mid + 1 ··· r]
 2 (here assume) L, R already sorted
 3 i ← 1, j ← 1     // two pointers for L and R
 4 ans ← 0      // total inversion number
  // let i, j iterator over two arrays
 5 while i ≤ mid − l + 1 and j ≤ r − mid do
   |  // looking for smallest L item larger than R
 6 |  if L[i] ≤ R[j] then
 7 |  |  i ← i + 1
 8 |  else
   |  |  // Found L[i] > R[j]!
   |  |  // then L[i] ··· L[mid − l + 1] each can form an inversion pair with R[j],
   |  |     remind here L subscript is diff from A, as stated above
   |  |  // so inversion num for R[j] is mid − l + 1 − i + 1 = mid − l − i + 2
 9 |  |  inv ← (mid − l − i + 2)
10 |  |  ans ← ans + inv
11 |  |  j ← j + 1
12 |  end
13 end
14 return ans
```

And, the whole algorithm for counting the inversion number will be:

Algorithm 9: Count-Inversion(A, l, r)

```
 1 if l = r then
 2 |  return 0
 3 end
 4 mid ← ⌊(l + r)/2⌋
 5 c1 ← Count-Inversion(A, l, mid)
 6 c2 ← Count-Inversion(A, mid + 1, r)
 7 MergeSort(A, l, mid)
 8 MergeSort(A, mid + 1, r)
 9 c3 ← Count(A, l, mid, r)
10 return c1 + c2 + c3
```

First call: Count-Inversion(A, 1, n).

So far, you may think this is an excellent algorithm since we only use $O(n)$ in each recursion step. However, it isn't! Remember, we have assumed each half is already sorted, but in fact they are random. If we firstly run some sort algorithm, say, Merge Sort, and then do the counting above, the whole running time will be:

$$T(n) = 2T(n/2) + \Theta(n \log n + n) = 2T(n/2) + \Theta(n \log n)$$

One can show $T(n) = \Theta(n \log^2 n)$.

This is, to a certain degree, acceptable, compared to previous $\Theta(n^2)$, but we still want to improve that. We can easily notice the main problem lies in sorting, which uses $\Theta(n \log n)$ in each recursion step. How can we reduce, or even avoid this process?

This is indeed hard to think about, but we can combine the sorting process (more concretely, Merge sort) with the process which we count inversion pairs that form between the two halves. In other words, previously we only do counting between two halves, now we also perform Merge at the same time. What will this lead to? Consider from recursion bottom(1 item), to top, each time we Merge the two halves, as what we did in Merge Sort, and at the same time, count inversion pairs that cross the two halves. And since we Merge from bottom to top, the two halves will always be sorted.(this is exactly the same Merge in Merge Sort)

The paragraph above is still so abstract, at least for myself, perhaps it's better to look at how the algorithm is implemented.

Algorithm 10: Merge-and-Count(A, l, mid, r)

```
   // same as previous algorithm, subscripts for L, R and A are different,
      remember this
 1 L ← A[l ··· mid], R ← A[mid + 1 ··· r]
 2 append ∞ at the end of L and R
 3 i ← 1, j ← 1        // two iteration pointers for L and R
 4 count ← 0        // counter for inversion number
 5 for k ← l to r do
 6 |   if L[i] ≤ R[j] then
 7 |   |   A[k] ← L[i]
 8 |   |   i ← i + 1
 9 |   else
10 |   |   A[k] ← R[j]
11 |   |   j ← j + 1
12 |   |   count ← count + (mid − l − i + 2)
13 |   end
14 end
15 return count
```

As you can find out above, apart from count inversion pairs between L and R, we merge them

into a new array A, this is exactly what we did in merge sort, which maintains the "sorted" invariant. With the function above, the complete algorithm for finding inversion number for an array is displayed below.

Algorithm 11: Sort-and-Count(A, l, r)

```
if l = r then
    return 0
end
mid ← ⌊(l + r)/2⌋
c1 ← Sort-and-Count(A, l, mid)
c2 ← Sort-and-Count(A, mid + 1, r)
c3 ← Merge-and-Count(A, l, mid, r)
return c1 + c2 + c3
```

First call: Sort-and-Count(A, 1, n)

2.5 The Maximum Subarray Problem

Problem: Given an array of size n, the task is to find the largest possible sum of a contiguous subarray. For example, given $[3, 2, 1, -7, 5, 2, -1, 3, -1]$, subarray $[5, 2, -1, 3]$ has the largest sum among all subarrays, we need to output $5 + 2 + (-1) + 3 = 9$.

We will provide a lot of algorithms to solve this problem.

2.5.1 brute force algorithm

The simplest idea is, for each pair (i, j), we calculate $A[i] + A[i + 1] + \cdots + A[j]$, and record maximum value we have seen along the process.

Algorithm 12: Max-Subarray-Brute-Force(A)

```
1  maxSum ← A[1]      // can also use -inf to initialize
2  for i ← 1 to n do
3  |   for j ← i to n do
   |   |   // calculate A[i] + ··· + A[j]
4  |   |   sum ← 0
5  |   |   for k ← i to j do
6  |   |   |   sum ← sum + A[k]
7  |   |   end
   |   |   // if current sum is larger, update maxSum
8  |   |   if sum > maxSum then
9  |   |   |   maxSum ← sum
10 |   |   end
11 |   end
12 end
13 return maxSum
```

This is a very simple algorithm, but requires $\Theta(n^3)$ running time.

2.5.2 prefix sum

In brute force algorithm, we notice that each time when we calculate $A[i] + A[i + 1] + \cdots + A[j]$, we need to iterate through these items, and add them together, which requires a lot of redundant work. The **prefix sum**, say $S[i]$, is defined as $\sum_{j=1}^{i} A[i]$, i.e., the sum of all items before(and include) $A[i]$. If we have a table of all $S[i]$ values, we can now rewrite $\sum_{k=i}^{j} A[k] = S[j] - S[i-1]$. See? That's a $\Theta(1)$ job!

Algorithm 13: Get-Prefix-Sum(A)

```
   // S[] records the prefix sum of array A
1  S[0] = 0
2  for i = 1 to n do
3  |  S[i] ← S[i − 1] + A[i]
4  end
5  return S
```

Algorithm 14: Max-Subarray-Prefix-Sum(A)

```
1  maxSum ← A[1]
2  S ← Get-Prefix-Sum(A)        // get prefix sum
3  for i ← 1 to n do
4  |  for j ← i to n do
5  |  |  sum ← S[j] − S[i − 1]        // calculate A[i] + ··· + A[j]
6  |  |  if sum > maxSum then
7  |  |  |  maxSum ← sum
8  |  |  end
9  |  end
10 end
11 return maxSum
```

This reduces the running time of our algorithm to $\Theta(n^2)$. (Calculating prefix sum only requires $\Theta(n)$, so overall $\Theta(n^2 + n) = \Theta(n^2)$)

2.5.3 divide and conquer

Again, we return to our topic, and again, we try to cut the array into two halves. Similar to **Inversion Number** example, we classified all subarrays into three cases:

1. entirely in the first half
2. entirely in the second half
3. crosses the cut

I think it will not surprise you that the third situation is the most difficult one, while the first two cases, can still be found recursively.

Algorithm 15: Max-Subarray-Divide-Conquer(A, l, r)

```
1  if l = r then
2  |  return A[l]
3  end
4  mid ← ⌊(l + r)/2⌋
5  max_1 ← Max-Subarray-Divide-Conquer(A, l, mid)
6  max_2 ← Max-Subarray-Divide-Conquer(A, mid + 1, r)
7  max_3 ← Max Subarray that crosses the cut
8  return max{max_1, max_2, max_3}
```

So how can we efficiently calculate max_3? Firstly, consider what do we mean by "crosses the cut"? That should be, the subarray will *at least include both* $A[mid]$ *and* $A[mid+1]$ in order to "cross". Hence these kind of subarray can always be divided into two parts: $A[i \cdots mid]$ and $A[mid+1 \cdots j]$ for some i and j. So in order to find max $A[i] + \cdots + A[j]$, we just find max $A[i] + \cdots + A[mid]$, and $A[mid+1] + \cdots + A[j]$, and finally add them together, this will definitely give us the max value.

Alright, so how can we find i?(and can use exactly the same method to find j) It should be the index that maximize $A[i] + \cdots + A[mid]$. This is much easier since one end, say, mid, is fixed. We initialize $maxSum$ to $-\infty$, then scan from mid towards left, each step we add an item to temporary sum, and update $maxSum$ if sum is larger. When we reach l(left end), we will have already iterated all possible indices i and stored the max sum in $maxSum$.

Algorithm 16: Max-Subarray-Divide-Conquer(A, l, r)

```
1  if l = r then
2  |  return A[l]
3  end
4  mid ← ⌊(l + r)/2⌋
5  max_1 ← Max-Subarray-Divide-Conquer(A, l, mid)
6  max_2 ← Max-Subarray-Divide-Conquer(A, mid + 1, r)
   // now let's count max_3
7  L_m ← −∞, R_m ← −∞
8  sum ← 0
9  for i = mid down to l do
10 |  sum ← sum + A[i]
11 |  if sum > L_m then
12 |  |  L_m ← sum
13 |  end
14 end
15 sum ← 0
16 for i = mid + 1 to r do
17 |  sum ← sum + A[i]
18 |  if sum > R_m then
19 |  |  R_m ← sum
20 |  end
21 end
22 return max{max_1, max_2, L_m + R_m}
```

First call **Max-Subarray-Divide-Conquer**(A, **1**, n).

It's not difficult to find out the process of finding max_3 requires $O(n)$ time, since we just scan throughout the array. If let $T(n)$ be the running time of whole algorithm, we will get:

$$T(n) = 2T(n/2) + O(n)$$

This gives $T(n) = O(n \log n)$.

2.5.4 linear time?

Review the idea of calculating max_3 above, we said that finding max $A[i] + \cdots + A[mid]$ is much easier since mid is a fixed ending point. This gives us an inspiration: for a *fixed* j, finding largest $A[i] + \cdots + A[j] = S[j] - S[i-1]$, is the same as finding the smallest $S[i-1]$.(Recall that $S[]$ is the prefix sum) If we can find the smallest $S[i-1]$ for each j, we will able to find the max subarray.(here $i-1$ must be strictly smaller than j, otherwise the subarray is null)

The process of finding smallest $S[i-1]$ can be easily done during the iteration through array. More concretely, we only need to update $minS = \min\{minS, A[i]\}$ at each step. Below shows the entire algorithm.

Algorithm 17: Max-Subarray-Linear(A)

```
   // Here we initialize minS to 0 because at least we can do S[j] − S[0] to
      ensure the sum is at least not smaller than S[j]
 1 maxSum ← −∞, minS ← 0
   // for each j, find minS, and then find S[j] − minS
 2 for j ← 1 to n do
   |  // update overall answer
 3 |  if S[j] − minS > maxSum then
 4 |  |  maxSum ← S[j] − minS
 5 |  end
   |  // update minS so far
 6 |  if S[j] < minS then
 7 |  |  minS ← S[j]
 8 |  end
 9 end
10 return maxSum
```

This algorithm can also be written without calculating prefix sum before, since each time we only use $S[j]$, we only need one variable to record $S[j]$ and accumulate it each time.

Algorithm 18: Max-Subarray-Linear2(A)

```
   // S will be all prefix sum so far, i.e., A[1] + ··· + A[j]
 1 maxSum ← −∞, minS ← 0, S ← 0
 2 for j ← 1 to n do
 3 |   S ← S + A[j]      // calculate prefix sum so far
 4 |   if S − minS > maxSum then
 5 |   |   maxSum ← S − minS
 6 |   end
   |   // update minS so far
 7 |   if S < minS then
 8 |   |   minS ← S[j]
 9 |   end
10 end
11 return maxSum
```

As you can see, this algorithm only requires linear $\Theta(n)$ time. It is indeed a difficult progress that we optimize the algorithm from $\Theta(n^3)$ down to $\Theta(n)$, with lots of new ideas come out. We say this is "More art than science".

2.5.5 dynamic programming

By using **dynamic programming** ideas, which we will formally introduce later, we can also design quite efficient algorithms, but efficient always requires more thinking. Here we just give you a first taste on DP.

We define $d[i]$ be the max sum of subarray that *ends with* $A[i]$. Here we must contain $A[i]$ in $d[i]$, otherwise, we cannot get $d[i+1]$ from $d[i]$, because it will break the "consecutive" subarray requirement. But if you ask me why we define in such a way, I cannot explain it, and that is the "art of dynamic programming" :)

So when we calculating $d[i]$, we only need to check $d[i-1]$ and $A[i]$, this is the basic idea of dynamic programming, that is, get value from previous values. And if $d[i-1] \leq 0$, which means $d[i-1] + A[i]$ is not larger than $A[i]$ itself! So why should we include $d[i-1]$ then, we just let $d[i] = A[i]$, this will be the max sum with $A[i]$ included. On the contrary, if $d[i-1] > 0$, then we should let $d[i] = d[i-1] + A[i]$, since include $d[i-1]$ makes the sum larger, and that is exactly what we want.

Algorithm 19: Max-Subarray-DP(A)

```
1  d[0] ← 0      // max sum of subarrays end with no item is 0
2  maxSum ← A[1]    // used to record max sum so far, notice here cannot
       initialize to 0
3  for i = 1 to n do
4  |  if d[i − 1] ≤ 0 then
5  |  |  d[i] ← A[i]
6  |  else
7  |  |  d[i] ← d[i − 1] + A[i]
8  |  end
9  |  if d[i] > maxSum then
10 |  |  maxSum = d[i]
11 |  end
12 end
13 return maxSum
```

This is also a $\Theta(n)$ algorithm, and as usual, it is not easy to think. But, we can still reduce the space it takes, i.e., space complexity. Notice here we use an array $d[i]$ to record the max sum of subarrays end with $A[i]$, but each time, say, when we calculate $d[i]$, we only use the previous one, say, $d[i-1]$. So it is no need that we use an array to track this: we only need a variable to record the previous d value, that's enough! So we can slightly modify the algorithm as below:

Algorithm 20: Max-Subarray-DP2(A)

```
1  previousD ← 0
2  maxSum ← A[1]
3  for i = 1 to n do
4  |  if previousD ≤ 0 then
5  |  |  previousD ← A[i]
6  |  else
7  |  |  previousD ← previousD + A[i]
8  |  end
9  |  if previousD > maxSum then
10 |  |  maxSum = previousD
11 |  end
12 end
13 return maxSum
```

2.6 The Master Theorem

2.6.1 Theorem and its proof

Theorem 2.6.1 The Master Theorem: Let $a \geq 1, b > 1, c \geq 0$ be constants, if $T(n) = aT(n/b) + n^d$, then:

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n), & \text{if } d = \log_b a \\ O(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

There is one kind of proof given in lecture slide, using expansion method. But personally, I like the method below. (Proof is not required in this course.)

Proof. Consider $T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$, for the 0th layer of recursion(since we haven't begun recursion), running time is $c \cdot n^d$; for the 1st layer, there are a branches, and each branch has running time $c \cdot \left(\frac{n}{b}\right)^d$, in total $c \cdot \left(\frac{a}{b^d}\right) \cdot n^d$; for the 2nd layer, each branch in 1st layer has a branches, so there are a^2 branches now, with each requires $c \cdot \left(\frac{n}{b^2}\right)^d$, and $c \cdot \left(\frac{a}{b^d}\right)^2 \cdot n^d$ in total. We can easily find the pattern: the running time of k-th layer is $c \cdot \left(\frac{a}{b^d}\right)^k \cdot n^d$.

Add all of them together, we get

$$T(n) = c \cdot n^d \cdot \left[1 + \left(\frac{a}{b^d}\right) + \cdots + \left(\frac{a}{b^d}\right)^k\right]$$

Recall the sum of geometric sequence:

$$1 + p + p^2 + \cdots + p^k = \begin{cases} k+1, & \text{if } p = 1 \\ \dfrac{p^{k+1} - 1}{p - 1}, & \text{if } p \neq 1 \end{cases}$$

Condition 1: when $a = b^d$, ratio is 1, so $T(n) = O(n^d \log n)$.

Condition 2: when $a < b^d$, the sequence is decreasing, so the sum is determined by the first item(you can also infer from the equation of sum above), then $T(n) = O(n^d)$.

Condition 3: when $a > b^d$, the sequence is increasing, the sum is determined by the last item, this gives $T(n) = n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = \left(n^{\log_n a}\right)^{\log_b n} = n^{\left(\frac{\ln a}{\ln n} \cdot \frac{\ln n}{\ln b}\right)} = n^{\log_b a}$. ■

2.6.2 equalities, inequalities and more

Notice that the theorem on previous page is quite restricted, it only involves the case when the form is like n^d, we cannot apply it to solve cases involving $\log n$, etc. Now we'd like to introduce more powerful versions of Master Theorem, but actually, when you understand the below cases, you will find that you only need to remember the previous simple versions, and I'll tell you why.

Theorem 2.6.2 The Master Theorem (for equalities):

If $T(n) = aT(n/b) + f(n)$, and denote $c = \log_b a$, then:

- If $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^c)$
- If $f(n) = \Theta(n^c)$, then $T(n) = \Theta(n^c \log n)$
- If $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq df(n)$ for some $d < 1$ and large enough n, then $T(n) = \Theta(f(n))$.

Then we can solve the following recurrence:

$$T(1) = 1,\ T(n) = 9T(n/2) + n^3 \log_2 n \text{ for } n > 1.$$

Notice that $c = \log_2 9 \approx 3.17$, hence $f(n) \leq n^{3.1} = O(n^{c-\epsilon})$, where $\epsilon \approx 0.07$. Thus $T(n) = \Theta(n^{\log_2 9})$. Here we use the fact that $\log n$ is smaller than any positive power of n.

Actually, this is the only case that we cannot use the previous "simple version Master Theorem" to solve recurrence, but since log is smaller than any power, you can simply regard it as nothing, i.e.,

$$T(1) = 1,\ T(n) = 9T(n/2) + n^3 \text{ for } n > 1.$$

and if you apply the "simple version Master Theorem", the result is the same, $T(n) = \Theta(n^{\log_2 9})$. See? You only need to remember the previous version.

You may notice in the third case above, there is one extra condition "if $af(n/b) \leq df(n)$ for some $d < 1$ and large enough n". This is called **Smoothness condition(regularity condition)**, and if you are interested, you can search on Google about why we need an extra condition. Fortunately, this condition satisfies most of the time(like the example below), there are only a few weird cases where this condition fails.

To show how we can check the **Smoothness condition**, we look at the example:

$$T(1) = 1,\ T(n) = 4T(n/2) + n^2\sqrt{n} \text{ for } n > 1.$$

Since $c = \log_2 4 = 2,\ f(n) = \Theta(n^{2.5}) = \Omega(n^{c+\epsilon})$ where $\epsilon = 0.5$, this is the third case, and we

need to check the regularity condition:

$$4 \cdot \left(\frac{n}{2}\right)^2 \sqrt{\frac{n}{2}} = \frac{\sqrt{2}}{2} \cdot n^2\sqrt{n} \leq d \cdot f(n)$$

for $d = \frac{\sqrt{2}}{2} < 1$ and large enough n. Therefore we can conclude that $T(n) = \Theta(n^2\sqrt{n})$.

There is also a version for inequalities:

Theorem 2.6.3 The Master Theorem (for inequalities):

If $T(n) \leq aT(n/b) + f(n)$, and denote $c = \log_b a$, then:

- If $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then $T(n) = O(n^c)$
- If $f(n) = O(n^c)$, then $T(n) = O(n^c \log n)$
- If $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq df(n)$ for some $d < 1$ and large enough n, then $T(n) = O(f(n))$.

There is nothing new above, we just replace "=" with "$\leq$" and changes the results to O rather than Θ.

2.7 Integer Multiplication

You may first think of using primary school method: i.e., "long multiplication", and this requires $\Theta(n^2)$ time. We will show that we can do better than this, but the ideas are quite difficult to think of, and actually people used quite a long time to invent the algorithms.

2.7.1 divide and conquer: first attempt

For example, we would like to calculate 3711×4021, we can divide each number into two parts: **high** part and **low** part, say, $x_h = 37, y_h = 40$, and $x_l = 11, y_l = 21$. Then, $x \times y = x_h \times y_h \cdot 10^n + (x_l \times y_h + x_h \times y_l) \cdot 10^{n/2} + x_l \times y_l$, where n is the length of two numbers. One thing worths mentioning is that we can always take n as a perfect square of 2, and if it is not, we just put some 0s in front of the number. So now, there are four multiplications and we can use recursion to calculate each of them. And for multiplying power of 10, we can just think of it as adding some 0s after the number, so this takes $O(n)$ time.

Algorithm 21: multiply-DC(A, B)

```
   // A[1···n] and B[1···n] are two arrays storing string of base 10.
      A[1], B[1] are least siginificant bits.(LSB)
 1 n ← size of A and B
 2 if n = 1 then
 3 |  return A[1] · B[1]
 4 end
 5 mid ← ⌊n/2⌋
 6 M1 ←multiply-DC(A[mid + 1···n], B[mid + 1···n])      // x_h × y_h
 7 M2 ←multiply-DC(A[1···mid], B[mid + 1···n])      // x_l × y_h
 8 M3 ←multiply-DC(A[mid + 1···n], B[1···mid])      // x_h × y_l
 9 M4 ←multiply-DC(A[1···mid], B[1···mid])      // x_l × y_l
   // Below we can put numbers in array directly, or append 0 at the end and
      add them together.  Assume res[] is filled with 0 at the beginning.
10 res[1···n] ← M4
11 res[mid + 1···] ← res[mid + 1···] + M2 + M3
12 res[n + 1···] ← res[n + 1···] + M1
13 return res
```

This will require a running time as $T(n) = 4T(n/2) + O(n)$, hence $T(n) = O(n^2)$, which doesn't improve our algorithm at all. One may think of using binary representation(base 2) instead of decimal, but this doesn't help either, though multiply by power of 2 can be done in $O(1)$ time, with the help of left shift(<<), write the result into the array still takes $O(n)$, regardless of the time converting an integer of base 10 into base 2. So basically, we need to reduce the time we call recursion to reduce the running time.

2.7.2 Karatsuba's method

As shown above, we need to calculating 4 multiplications, which makes us to call 4 recursions. Trying to improve that, Karatsuba noticed that we only need $x_h \times y_l + x_l \times y_h$(the sum), instead of calculating each of the multiplication result. He suggested we only need to calculate 3 times, and they are: $M_1 = x_h \times y_h, M_2 = x_l \times y_l, M_3 = (x_h + x_l) \times (y_h + y_l)$, then we can get $x_h \times y_l + x_l \times y_h$ by doing $M_3 - M_1 - M_2$. This successfully reduce the running time of multiplication algorithm, with only 3 recursion calls: $T(n) = 3T(n/2) + O(n)$, gives $T(n) = n^{\log_2 3} \approx n^{1.585}$.

Algorithm 22: Karatsuba(A, B)

```
// A[1···n] and B[1···n] are two arrays storing string of base 10.
   A[1], B[1] are LSB.
```
$n \leftarrow$ size of A and B
if $n = 1$ **then**
  return $A[1] \cdot B[1]$
end
$mid \leftarrow \lfloor n/2 \rfloor$
$M_1 \leftarrow$Karatsuba($A[mid+1 \cdots n]$, $B[mid+1 \cdots n]$) // $x_h \times y_h$
$M_2 \leftarrow$Karatsuba($A[1 \cdots mid]$, $B[1 \cdots mid]$) // $x_l \times y_l$
$A' \leftarrow A[mid+1 \cdots n] + A[1 \cdots mid]$
$B' \leftarrow B[mid+1 \cdots n] + B[1 \cdots mid]$
$M_3 \leftarrow$Karatsuba(A', B') // $(x_h + x_l) \times (y_h + y_l)$
```
// Assume res[] is filled with 0 at the beginning.
```
$res[1 \cdots n] \leftarrow M_2$
$res[mid+1 \cdots] \leftarrow res[mid+1 \cdots] + M_3 - M_1 - M_2$
$res[n+1 \cdots] \leftarrow res[n+1 \cdots] + M_1$
return res

2.7.3 So far...

Inspired by Karatsuba, people can improve his algorithm by "dividing each integer into 3 parts, and solve 5 multiplications", or "divide into n parts, and solve $2n-1$ multiplications" etc. Later on, in 1971, Strassen solved this problem in $O(n \log n \log\log n)$, using **Fast Fourier Transformation(FFT)**. In 2007, $O(n \log n \cdot 8^{\log^* n})$ algorithm was found and in 2019, finally, $O(n \log n)$ algorithm was found.

However, Karatsuba's algorithm isn't always faster than our primary school $O(n^2)$ method, since it has a larger constant. In practice, people find that for integers with length less than 20, using our $O(n^2)$ method is better; while Karatsuba's algorithm is better for length $20 \sim 2000$, FFT is better for > 2000. And for your reference, Python uses 70 as a critical value to judge whether to perform primary school method or Karatsuba's method.

2.8 Matrix Multiplication

Given two $n \times n$ matrices A, B, how can we compute $C = AB$?

Since $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, we can use three nested loops to calculate each items in C, in $\Theta(n^3)$ time. This is our brute force algorithm.

2.8.1 divide and conquer?

Much similar to integer multiplication, we try to divide A and B into $\frac{1}{2}n \times \frac{1}{2}n$ matrices and call recursion to multiply each part.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

and,

$$C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \qquad C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22})$$
$$C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \qquad C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22})$$

This algorithm requires $T(n) = 8T(n/2) + O(n^2)$, notice here add matrices is $O(n^2)$. We can easily know $T(n) = O(n^3)$, from Master's Theorem.

2.8.2 Strassen's method

This is not easy to improve. But inspired by integer multiplication, Strassen managed to calculate that with only 7 multiplications:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

And this reduce the running time down to $O(n^{\log_2 7}) \approx O(n^{2.807})$.

Again, many people are trying to reduce the time complexity and another competition arose. We would not go into details here.

3. Randomized Algorithm

3.1 Recap: Probability

Here are some commonly used definitions.

Expectation: $\mathbb{E}(X) = \sum i \cdot \Pr(X = i)$

Linearity of expectation: $\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$, no matter X and Y are independent or not.

Indicator random variables: X only takes 0 or 1, which means $\mathbb{E}(X) = \Pr(X = 1)$.

Example 1: coin comes up heads with probability p and tails with $1 - p$. Find the expectation of flips X until first head is seen.

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{j=1}^{\infty} j \cdot \Pr(X = j) \\
&= \sum_{j=1}^{\infty} j \cdot (1-p)^{j-1} \cdot p \\
&= \frac{p}{1-p} \sum_{j=1}^{\infty} j \cdot (1-p)^{j} \\
&= \frac{p}{1-p} \cdot \frac{1-p}{p} = \frac{1}{p}
\end{aligned}$$

The last step is somehow mysterious, the brief idea is:

$$\left(\sum_{n=1}^{\infty} x^i\right)' = \sum_{n=1}^{\infty} i \cdot x^{i-1}$$

multiply each side by x, and notice the left hand side is the derivative of geometric series, then:

$$x \cdot \left(\frac{x}{1-x}\right)' = \sum_{n=1}^{\infty} i \cdot x^i$$

This gives the last step above.

Example 2: Roll two dice. What is the expected total value X?

It is trivial that $\mathbb{E}(X_1) = \mathbb{E}(X_2) = 3.5$, then:

$$\mathbb{E}(X_1 + X_2) = \mathbb{E}(X_1) + \mathbb{E}(X_2) = 7$$

3.2 The Hiring Problem

Consider we're looking for an assistant and there are n candidates. We would like to hire the best one, so we interview one by one, and if the current one is better than the best one we've seen before, we just fire the previous one and hire the current one.

Algorithm 23: Hire-Assistant(n)

```
1 best ← 0
2 for i ← 1 to n do
3 |   interview candidate i
4 |   if candidate i is better than best then
5 |   |   fire best
6 |   |   hire candidate i
7 |   |   best ← i
8 |   end
9 end
```

This algorithm runs fine, but there is one problem that we may hire too many people(worst case n) before we finally find the best one, so it may cost lots of money to fire old ones. In order to

make things better, we consider interview the candidates *in a random order*.

Algorithm 24: Hire-Assistant(n)

```
1  randomly permute all n candidates
2  best ← 0
3  for i ← 1 to n do
4  |   interview candidate i
5  |   if candidate i is better than best then
6  |   |   fire best
7  |   |   hire candidate i
8  |   |   best ← i
9  |   end
10 end
```

So what is the expected number of hires in the algorithm above? Let an indicator variable X_i where:

$$X_i = \begin{cases} 1 & \text{,if we hire candidate } i \\ 0 & \text{,if we don't} \end{cases}$$

Then apparently the number of hires $X = X_1 + \cdots + X_n$, thus the expected number of hires, $\mathbb{E}(X) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_n)$ according to the linearity of expectation.

Then what is $\mathbb{E}(X_i)$? Since X_i is an indicator variable, $\mathbb{E}(X_i) = \Pr(X_i = 1)$. We hire the candidate i if and only if he/she is the best among all first i candidates, and since they are arranged randomly, the probability that the best one among first i candidates is at the last position is $\frac{1}{i}$. Thus, $\mathbb{E}(X_i) = \frac{1}{i}$, and

$$\mathbb{E}(X) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n} = \Theta(\log n)$$

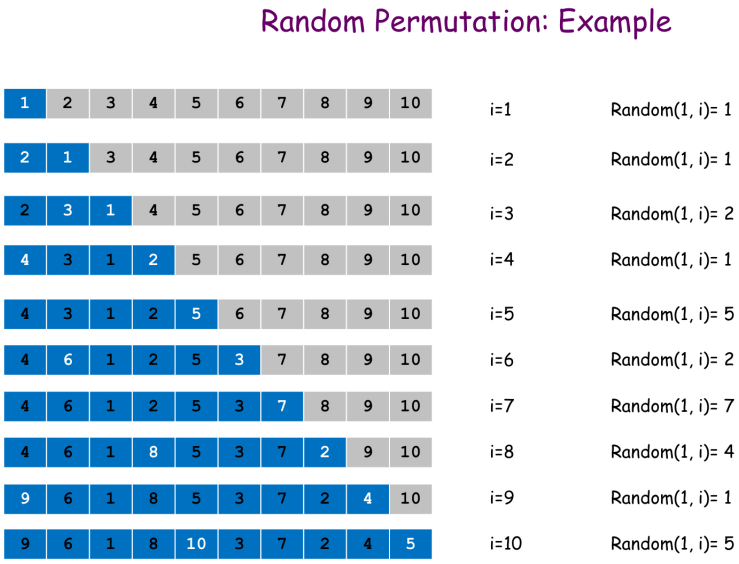
3.3 Generating a random permutation

Notice that in the hiring problem above, we first need to randomly permute those candidates. But how can we do that? (Here randomly means all permutations, in total $n!$, should appear with equal probability $\frac{1}{n!}$)

Let's first look at the implementation of this algorithm, and then explain why it can perform the job well. Assume out computer has a procedure $Random(i, j)$ that can generates a **random uniform** integer between i and j. (**uniform** means each int occurs with same probability)

Algorithm 25: RandomPermute(A)

```
1 n ← A.length
2 for i ← 1 to n do
3 |   swap A[i] with A[Random(1, i)]
4 end
```



In the example above, take the last step as an example, the random number we generated is 5, so we will swap the item at index 5, which is 5, with the last item 10. Then 10 ends up with index 5 and 5 ends up with index 10.

Notice this process is actually *revertible*, from the end to beginning. For example, we notice at the end, $A[5] = 10$, so the last step we must have swapped $A[5]$ with $A[10]$, since before last step, $A[10] = 10$. Then if we "revert" this step, i.e., swap $A[5]$ and $A[10]$, then the array will back to the status before last step.

Then, if we denote the random number generated at step i to be r_i, the n-tuple $(r_1, \cdots, r_n)$ will generate a permutation of the array. Moreover, given the result permutation, we can actually "revert" the process, like we discussed above, to get the $(r_1, \cdots, r_n)$ generated. This means, a tuple $(r_1, \cdots, r_n)$ is **uniquely corresponding** to a permutation at last. So if two tuples are different, the permutations they generate are different as well!

Now we can easily prove the correctness of this algorithm, since each tuple is generated with probability $\frac{1}{n!}$, (each r_i is generated with probability $\frac{1}{n}$), then each permutation is also generated with probability $\frac{1}{n!}$.

Here we will give another proof by induction, to show that "after the i-th iteration, $A[1 \cdots i]$ has been randomly permuted."

Base case: $i = 1$, trivial.

Inductive step: Assume $A[1 \cdots i-1]$ has been randomly permuted after $i-1$ iterations of the algorithm, then we will calculated the probability that $A[1 \cdots i] = (a_1, \cdots, a_i)$ appears after the

i-th iteration.

For example, if $i = 10$, then after $(i-1)$ steps, the array must be something like this:

$$[2, 8, 7, 3, 4, 1, 5, 6, 9, 10]$$

then what is the probability that we generate $[2, 8, 10, 3, 4, 1, 5, 6, 9, 7]$ after i-th step? We must swap $A[3] = 7$ with $A[10] = 10$, which means $Random(1, i)$ must return 3 to achieve that.

To summarize the above paragraph, we can get $A[1 \cdots i] = (a_1, \cdots, a_i)$ after i-th iteration *if and only if*

- $Random(1, i)$ returns a specific number and
- $A[1 \cdots i-1]$ is a specific $(a_1, \cdots, a_{i-1})$

The second point above has a probability $\frac{1}{(i-1)!}$ according to assumption of induction, the first point above has a probability $\frac{1}{i}$. Hence, the probability that $A[1 \cdots i] = (a_1, \cdots, a_i)$ is $\frac{1}{(n-1)!} \cdot \frac{1}{i} = \frac{1}{i!}$, which means that it's a random permutation.

3.4 Quick Sort

Quick Sort is somehow similar to **Merge Sort**. Instead, it chooses a **pivot** each time, and then **partitions** the array so that all items less than or equal to the pivot are on the left and all items greater than pivot are on the right.

Then it recursively calls QuickSort to left and right sides.

Algorithm 26: QuickSort(A, l, r)

```
1 if l ≥ r then
2 |   return
3 end
4 pivot ← Partition(A, l, r)
5 QuickSort(A, l, pivot − 1)     // quick sort left part
6 QuickSort(A, pivot + 1, r)     // quick sort right part
```

For the **Partition** process, we need to choose a pivot first, here we simply choose the last element as pivot. Then we divide the array $A[l \cdots r]$ into two parts: $A[l \cdots i]$ are all smaller or equal than pivot $A[r]$, $A[i+1 \cdots j-1]$ are all larger than pivot $A[r]$, while $A[j \cdots r-1]$ are not decided yet.

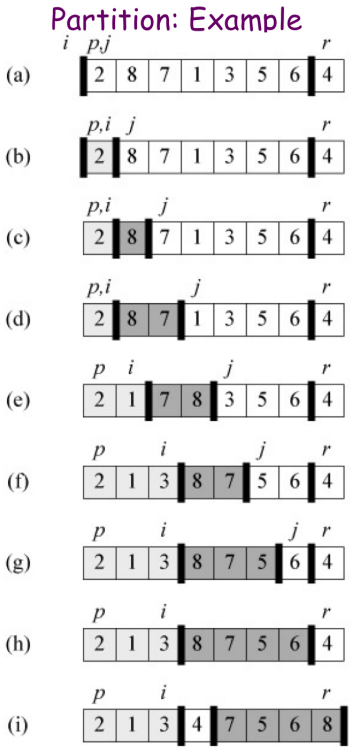
To divide, we perform "swaps": iterate j from l to $r-1$, if $A[j]$ is smaller or equal than pivot $A[r]$, we expand the area of left part, i.e., $i \leftarrow i+1$, and then swap $A[j]$ with $A[i]$. This will enlarge smaller subpart by size 1, and put the newly found item into that part. On the contrary,

if $A[j]$ is larger than pivot, then we just expand the larger subpart by size 1, say $j \leftarrow j+1$, and no need to do other stuff since $A[j]$ will be already contained after we expand the part.

Here is the implementation of this process:

Algorithm 27: Partition(A, l, r)

```
1  x ← A[r]      // choose last item to be pivot
2  i ← l − 1     // No item found yet, so there should be nothing in A[l···i]
3  for j ← l to r − 1 do
4  |  if A[j] ≤ x then
5  |  |  i ← i + 1      // expand left part
6  |  |  swap A[i] and A[j]      // include the new item into left part
7  |  end
8  end
9  swap A[i + 1] and A[r]      // make pivot to be at middle
10 return i + 1      // return pivot
```



Now we would like to analyze the running time of Quick Sort. Firstly, for the best case, where we *can always select median element as pivot*, we can divide the array into two parts every time, which gives $\Theta(n \log n)$. However, for the worst case, for example, if we always select the smallest(or largest) element as pivot, then it will be $\Theta(n^2)$.

Thus, in reality, we **randomly** choose a pivot in the array, and this is **randomized algorithm**: making a random choice each time it chooses a pivot.

For quick sort, or a randomized algorithm, we often care about its **expected running time**, denote as $\mathbb{E}[T(I,R)]$, where I is the input(the array in quick sort), while R is a random string or numbers that decide what we choose as random each time.

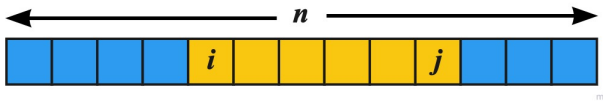
Without loss of generality, we can assume all elements are distinct, since if this is not the case, we can regard original input $A[1\cdots n]$ as $B[1\cdots n]$, where $B[i]=\{A[i],i\}$. Then all elements in B are distinct and we can just manipulate on $B[]$ instead. Also, remember the running time is *proportional to* number of comparisons. Notice that for any two items, they will either not be compared, or be compared only once, that is when one of them is the pivot. Since any two items can *at most compare once*, we can define indicator random variable:

$$X_{ij}=\begin{cases}1 & ,\text{if } i-\text{th smallest item is ever compared with } j-\text{th smallest item}\\ 0 & ,otherwise\end{cases}$$

Then,

$$\begin{aligned}\mathbb{E}(runtime) &\leq \mathbb{E}\left(c\cdot\left(\sum_{i<j}X_{ij}\right)\right)\\ &= c\cdot\sum_{i<j}\mathbb{E}(X_{ij})\\ &= c\cdot\sum_{i<j}\Pr(X_{ij}=1)\end{aligned}$$

So now we consider how to find $\Pr(X_{ij}=1)$, that is, the probability of i-th smallest and j-th smallest elements are ever compared. (In the image below, assume items are arranged in ascending order)



In the image above, we divide the array into two color regions, where between i and j(including i and j) are colored with yellow.

- if pivot is in **blue region**, then i and j will be then allocated to the same subpart of array, during this process, they cannot be compared to each other.
- if pivot is in **yellow region**, then this level is the last chance for i and j to be compared. And moreover, i and j will be compared *if and only if* i or j is chosen to be the pivot this level.

Thus, either they will be compared or not is *decided at the level which pivot is chosen in yellow region.* And at that level, they will be compared if and only if one of them is chosen as the pivot.

Therefore,

$$\Pr(X_{ij} = 1) = \frac{2}{j - i + 1}$$

Now we can calculate:

$$\mathbb{E}(runtime) = c \cdot \sum_{i<j} \mathbb{E}(X_{ij}) = c \cdot \sum_{i=1}^{n-1} \sum_{j=2}^{n} \frac{2}{j - i + 1}$$

The summation is hard to calculate directly, but let's make a table:

When $i = 1$, $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}$

When $i = 2$, $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-2} + \frac{1}{n-1}$

When $i = 3$, $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-2}$

$\cdots$

When $i = n - 1$, $\frac{1}{2}$

To sum up *by column*, we get:

$$(n-1) \cdot \frac{1}{2} + (n-2) \cdot \frac{1}{3} + (n-3) \cdot \frac{1}{4} + \cdots + 1 \cdot \frac{1}{n}$$

And with some tricks:

$$\begin{aligned}
&(n-1) \cdot \frac{1}{2} + (n-2) \cdot \frac{1}{3} + (n-3) \cdot \frac{1}{4} + \cdots + 1 \cdot \frac{1}{n} \\
\leq\ & n \cdot \frac{1}{2} + n \cdot \frac{1}{3} + n \cdot \frac{1}{4} + \cdots + n \cdot \frac{1}{n} \\
=\ & n \cdot \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n} \right) \\
\leq\ & n \cdot \int_1^n \frac{1}{x}\, dx = n \ln n
\end{aligned}$$

Despite the constant c, the expected running time for quick sort is still $\Theta(n \log n)$.

3.5 Randomized Selection

Problem: Given an array A of n distinct elements, found the i-th smallest element in A.

II Sorting Algorithms

4. Other Basic Sorting Algorithms (optional)

Notice: Insertion sort, merge sort and quick sort have been covered in Topic 00, 02, 03, respectively, we will not cover them in this note.

These two algorithms are not paid much attention during this course. Selection Sort is introduced in first class which is designed to give students an intuition of algorithm, while Bubble Sort is introduced in Tutorial session.

4.1 Selection Sort

This is quite a natural algorithm: to sort an array of elements, firstly, you find the smallest item in the array, put it to first position, and find the smallest item in remaining of the array(i.e., except for the first element), and put it to the second position.

The real implementation of Selection Sort is just like this, only one thing to amend is that when we find the smallest one and put it to first position, what should we do to the original item that at 1st position? There are two different methods:

1. We put sorted items into a new array, i.e., each time we find the smallest item, we insert it into the new array, so the new array will be sorted
2. Or, we directly swap the two items. For example, if the smallest item is at position 4, we swap $A[1]$ and $A[4]$. This will not affect our later processes.

Apparently, first method is simple but requires extra **space complexity** (remember this word?),

so usually we choose the second one.

Let's have an example for this algorithm:

- to sort array: $(5, 2, 8, 6, 1)$
- 1st step: find the smallest one, 1, swap with first item, results in $(1, 2, 8, 6, 5)$
- 2nd step: find the smallest except for first item, 2, swap with second item, results in $(1, 2, 8, 6, 5)$
- 3rd step: find the smallest except for first two items, 5, swap with third item, results in $(1, 2, 5, 6, 8)$
- $\cdots$

The pseudocode should be like this:

Algorithm 28: Selection-Sort-1($A[1 \cdots n]$)

```
1  for i ← 1 to n − 1 do
       // Now we want to find the smallest item among A[i ··· n]
2      minNum ← A[i]      // Records the smallest item we've met
3      minIndex ← i       // Records the index of minNum
4      for j ← i + 1 to n do
           // If a smaller one found, record it!
5          if A[j] < minNum then
6              minNum ← A[j]
7              minIndex ← j
8          end
9      end
       // After finding the smallest one, swap with A[i]
10     swap A[i] and A[minIndex]
11 end
```

The lecture slides use a different implementation, instead of looking for the smallest one and, at the end, swap it with current one(i-th item), we continuously swap $A[i]$(current one) and $A[j]$ whenever we find such an $A[j]$ that smaller than $A[i]$. For example, to sort $(5, 2, 8, 6, 7, 1)$,

- First $i = 1$, $A[i] = 5$, and we loop j from 2 to 6, to check if there is a smaller item
- when $j = 2$, $A[j] = 2 < A[i] = 5$, we swap them, results in $(2, 5, 8, 6, 7, 1)$
- when $j = 3$, $A[j] = 8 > A[i] = 2$, don't swap
- when $j = 4$, $A[j] = 6 > A[i] = 2$, don't swap
- when $j = 5$, $A[j] = 7 > A[i] = 2$, don't swap
- when $j = 6$, $A[j] = 1 < A[i] = 2$, swap them, results in $(1, 5, 8, 6, 7, 2)$

The pseudocode of this algorithm looks much shorter, but actually their thoughts and time complexity don't differ too much.

Algorithm 29: Selection-Sort-2($A[1 \cdots n]$)

```
1 for i ← 1 to n − 1 do
2 |   for j ← i + 1 to n do
  |   |   // Look for smaller item, and swap with A[i]
3 |   |   if A[i] > A[j] then
4 |   |   |   swap A[i] and A[j]
5 |   |   end
6 |   end
7 end
```

The correctness of **Selection Sort** can be proved by induction.

Theorem 4.1.1 When **Selection Sort** terminates, the array is sorted.

Proof. We use induction to prove it.

Basic step: When $n = 1$, the algorithm is obviously correct.

Inductive step: Assume the algorithm sorts *every array of size* $n-1$ correctly, then for an size n array $A[1 \cdots n]$, what the algorithm does is that:

- It first find the smallest item, and put it at $A[1]$
- Then, it omits the first item, and runs **Selection Sort** on $A[2 \cdots n]$: where by induction, this correctly sort $A[2 \cdots n]$
- Since $A[1]$ is the smallest, the whole array $A[1 \cdots n]$ are sorted.

■

The running time of Selection Sort is easy to analyze. Take the second algorithm above as example, line 3 and line 4 will run for each iteration of the two loops, so the running time is

$$\sum_{i=1}^{n-1} \sum_{i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

which immediately follows that the running time of Selection Sort is always $\Theta(n^2)$.

4.2 Bubble Sort

The idea of **Bubble Sort** is that: each time we check two adjacent items, if the left one is larger, we swap them. Then, after one iteration through the whole array $A[1 \cdots n]$, the largest item will be "bubbled" to right-most position. Then, we continuously do this on $A[1 \cdots n-1]$(except

for the last one, which already sorted), and now the second-largest item will be "bubbled" to second-rightmost position.

The pseudocode is like this:

Algorithm 30: Bubble-Sort-1($A[1 \cdots n]$)

```
1 for i ← 1 to n do
      // Could you explain why j terminates at n − i?
2     for j ← 1 to n − i do
          // Check adjacent, if left is larger, swap them
3         if A[j] > A[j + 1] then
4             swap A[j] and A[j + 1]
5         end
6     end
7 end
```

This algorithm looks stupid, since it requires lots of "checking and swaps". Consider the case where the input is already sorted, the **Bubble Sort** algorithm above still perform ALL two nested loops, and check each adjacent items to see if they need swaps. Inspired by this, we can directly terminates the algorithm if *the previous j iteration does zero swaps*. If the j loops from begin to end and find that no swap is required, then no surprising the array must have already been sorted.

So now we can change the algorithm into:

Algorithm 31: Bubble-Sort-2($A[1 \cdots n]$)

```
1 swapped ← true     // Record whether we swapped last time, here initialize
      to true
2 while swapped is true do
3     swapped ← false for i ← 1 to n − 1 do
4         if A[i] > A[i + 1] then
5             swap A[i] and A[i + 1]
6             swapped ← true
7         end
8     end
9 end
```

The correctness of Bubble Sort can be obtained by the correctness of below two theorems:

Theorem 4.2.1 When Bubble Sort terminates, the array must be sorted.

Proof. The algorithm terminates only if the last pass did not swap any pair, i.e

$$A[1] \leq A[2] \leq \cdots \leq A[n-1] \leq A[n],$$

which means that the array is sorted. ■

Theorem 4.2.2 After the i-th pass, the items in locations $A[n-i+1, \cdots, n]$ are in their proper sorted locations and none of them ever move again.

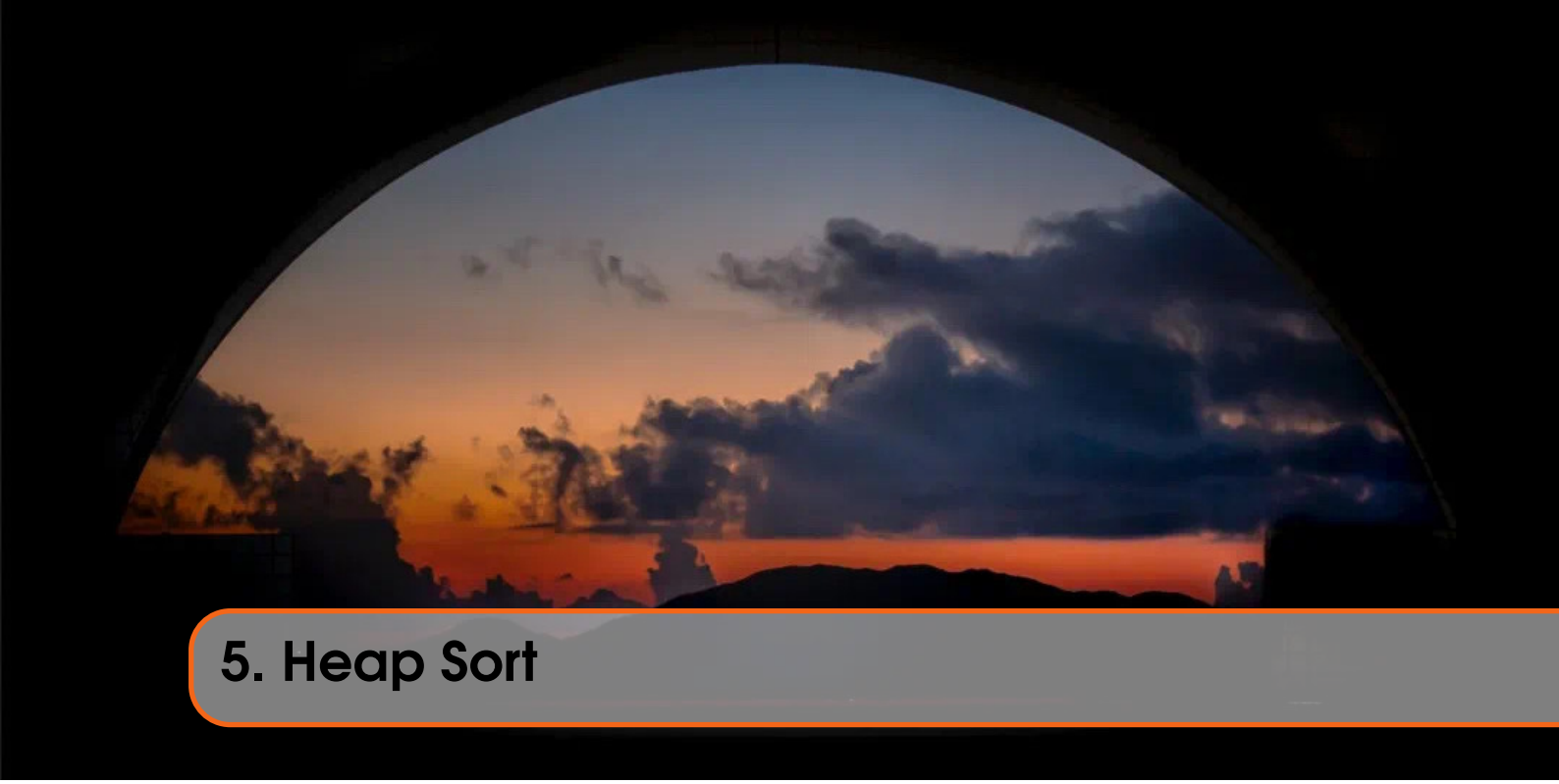
Proof. We prove this claim by induction.

- This is obviously true if $n = 1$ or $n = 2$.
- After the 1st pass, the largest element must be in location $A[n]$, and that item never moves after the first pass completes.
- After the 1st pass completes, it is as if Bubble Sort is being run on array $A[1 \cdots n-1]$.
- The claim then follows by induction.

■

For time complexity, the best case is when the array is already sorted, $O(n)$, while the worst case is when the array is reversely sorted, $O(n^2)$, the worst case complexity can be directly observed from our first Bubble Sort algorithm.

There is also a formal proof of worst case in problem set SS6. I'd like to omit details here.



5. Heap Sort

5.1 Intro. to Heap

Omitting excessive introductions, now we want a data structure that supports two operations:

- **Insert**: insert a new element into it
- **Extract-Min**: removes and returns the smallest element inside

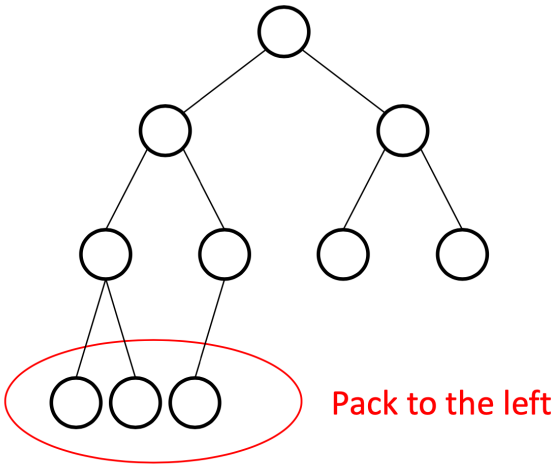
For now, you may come up with some possible implementations:

- Use a list:
  - **Insert** requires $O(1)$ time.
  - **Extract-Min** requires $O(n)$ time, since you need scan through the list to find the smallest one
- Use a sorted list:
  - **Insert** requires $O(n)$ time, since in order to maintain the list to be sorted, you need to scan through the list to find a suitable place to insert the new element.(like insertion sort)
  - **Extract-Min** requires $O(1)$ time, since list is sorted.

However, we are not satisfied with the results above, let's introduce **Heap**.

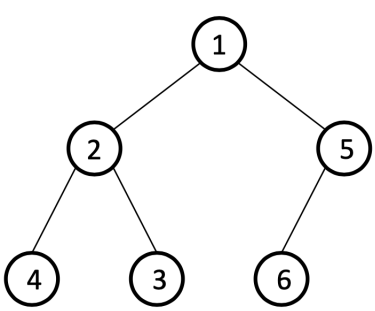
Definition 5.1.1 **Heaps** are **binary trees**, with the restrictions:

- All levels must be full, except for the lowest level
- If the lowest level is not full, all nodes must be packed to the left.

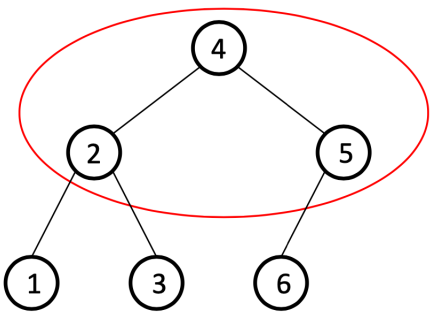


We also define a special kind of heap called **Min-heap**.

Definition 5.1.2 **Min-heap** is a **heap** in which the value of a node is *at least* the value of its parent.



A min-heap



not a min-heap

Since **heap** is a special kind of **binary tree**, its height h and number of elements n are related:

Theorem 5.1.1 For a **heap** with n elements and height h, there must be

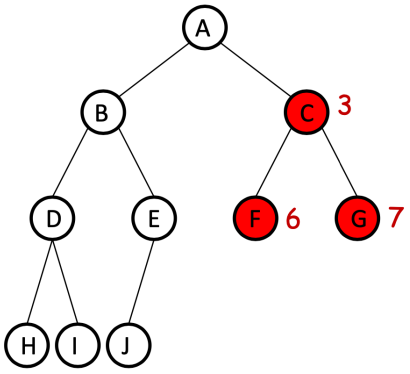
$$2^h \leq n < 2^{h+1}$$

thus an n-element heap has height $\Theta(\log n)$.

Proof. The proof is trivial, only use the fact that a binary tree with height h can have at most

$2^{h+1} - 1$ nodes(height starts from 0), so a heap with height h must have $2^h - 1$ nodes before level h(since they must be full). ■

Interestingly, the structure of **heap** is so regular so that we can use an array to represent it.



1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

Example: C = $A[3]$.
C's children are
$F = A[2 \cdot 3] = A[6]$ and $G = A[2 \cdot 3 + 1] = A[7]$.
G's parent is C = $A[3] = A[\lfloor 7/2 \rfloor]$.

- The root is in array position 1
- For any element in array position i
  - The left child is in position $2i$
  - The right child is in position $2i + 1$
  - The parent is in position $\lfloor i/2 \rfloor$

Now with the properties of heap, we can introduce how we can use heap to efficiently perform **insertion** and **extract-min** operations.

5.2 Insertion

6. Linear Sort

6.1 Lower Bound for Comparison Sorts

So far, all sorting algorithms we have seen perform no better than $\Omega(n \log n)$. You may wonder if it is possible to have a much faster sorting algorithm, and the answer is No, if we *restricted to comparison sorts*, i.e., each time we compare two numbers and decide some kind of order of them. All sorting algorithm we have seen are **comparison-based** sorts, and we can prove that any sorting algorithm based only on comparison cannot perform better than $\Omega(n \log n)$.

Theorem 6.1.1 Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

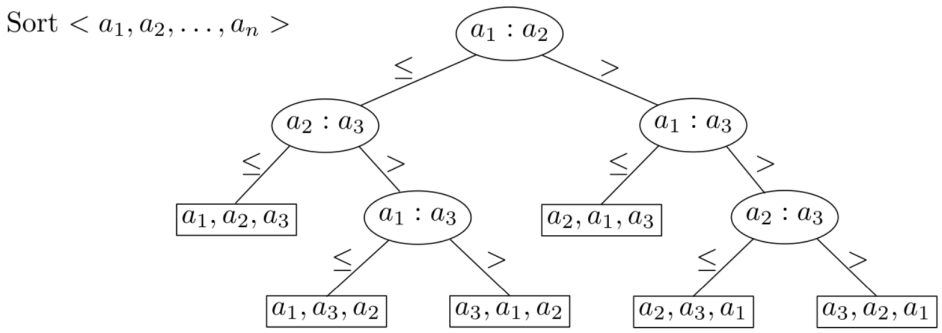
Note that the correctness of the theorem above directly gives "$\Omega(n \log n)$ is a lower bound for all **Comparison-Based Sorting** algorithms."

To proof the theorem, we first introduce **Decision Tree**, with which we can view the process of comparison sorts.

Definition 6.1.1 A **decision tree** is a *full binary tree* that represents the comparisons between elements that are performed by a particular sorting algorithm on an input of given size.

In a decision tree, each *non-leaf* node is represented by $i : j$ to indicate that we are comparing i and j, and if $i \leq j$, we go to the left subtree, while $i > j$ goes to the right. Each *leaf* node represents a permutation of the result of sorting algorithm, corresponding to an order of all items.

The example below can help you have a better understanding of decision tree.



So the decision tree above basically imitates the process of our comparison sort, for example, we first ask "Is a_1 larger than a_2?", assuming the answer is No, we go to left subtree, and then ask "Is a_2 larger than a_3?", and if the answer is also No, we go to left subtree again, and reach a leaf node, which means now we can determine their order, which is $a_1 \leq a_2 \leq a_3$.

Notice that any correct sorting algorithm must be able to produce *each permutation* of its input, each of the $n!$ permutations of a size n input must appear as one of the leaves of the decision tree for a comparison sort to be correct. (think why, though this is intuitive)

Therefore, the decision tree must have $n!$ leaves since a sorting algorithm can have $n!$ different outputs for each possible permutation on n items. Moreover, a binary tree(recall decision tree is binary) with $n!$ leaves must have height $\Omega(\log n)$(see theorem below). Therefore, the height of tree must be

$$\Omega(\log(n!)) = \Omega(n \log n)$$

which means we need $\Omega(n \log n)$ comparisons in this algorithm, therefore comparison-based sorting algorithm requires at least $\Omega(n \log n)$ time.

Theorem 6.1.2 A binary tree with n leaves must have height $\Omega(\log n)$.

Proof. Firstly we know a binary tree of height h has at most 2^h leaves. (This should follow immediately from the definition of binary tree) Since the tree has n leaves, we have

$$n \leq 2^h \quad \Rightarrow h \geq \log_2 n$$

■

6.2 Counting Sort

So, it is true that we cannot do better for sorting? No! We may consider other sort algorithms that doesn't depend on comparisons.

Here is one intuitive method. Consider you have n integers and all of them are between 1 and k. For example, if $n = 5$ and $k = 4$, you may have:

$$a_1 = 4 \quad a_2 = 2 \quad a_3 = 1 \quad a_4 = 4 \quad a_5 = 2$$

Now consider you have k "buckets", labelled from 1 to k, whenever you meet an item x, you put it into the bucket with label x. Now your buckets should look like:

Bucket	1	2	3	4
# of items in the bucket	1	2	0	2

because you have one 1, two 2s and two 4s.

Now if you have those "buckets", it will be easy for you to sort those numbers, for example, a_5 should be put at new position 3, since there are one item in bucket 1, and two items in bucket 2(one of the two items is a_5 itself) that are less than or equal to a_5. So the new order will be:

Order after sorting	1	2	3	4	5
Item			$a_5 = 2$		

Notice if we first check a_2, then it will also be put at new position 3, for the same reason above. However, we would not do that because it will lead to *unstable sort*. Notice that $a_2 = a_5 = 2$ at the beginning, so if the sort is *stable*, we still want a_2 to be in front of a_5 after sorting. Inspired by this, we should *check items backwards*, i.e., check a_5, then a_4, then a_3, a_2, a_1. This guarantee our algorithm to be *stable*. (please think about this point twice, it is important for you to know why check backwards can make it stable)

Back to our sorting process, remember we check items *backwards*, so now we check a_4. Similarly, a_4 should be put at new position 5, since there are one item in bucket 1, two items in bucket 2 and two items in bucket 4(one of the two items is a_4 itself) that are less than or equal to a_4.

Order after sorting	1	2	3	4	5
Item			$a_5 = 2$		$a_4 = 4$

Then a_3, there is only one item in bucket 1, which is a_3 itself, so it should be put at position 1.

Order after sorting	1	2	3	4	5
Item	$a_3 = 1$		$a_5 = 2$		$a_4 = 4$

Then a_2, notice though there are one item in bucket 1, and two items in bucket 2(one of the two items is a_2 itself) that are less than or equal to a_2, we have *already ejected* a_5, so now it should be put at position $1 + 1 = 2$ rather than $1 + 2 = 3$. (think twice, make sure you understand the process)

Order after sorting	1	2	3	4	5
Item	$a_3 = 1$	$a_2 = 2$	$a_5 = 2$		$a_4 = 4$

Then a_1, similar to a_2, it should be put at position $1 + 2 + 1 = 4$ rather than $1 + 2 + 2 = 5$ since a_4, which equals to a_1, has already been ejected.

Order after sorting	1	2	3	4	5
Item	$a_3 = 1$	$a_2 = 2$	$a_5 = 2$	$a_1 = 4$	$a_4 = 4$

Now the sorting process terminates, with order a_3, a_2, a_5, a_1, a_4, with no surprise, this is a *stable sort algorithm*.

Remark: After the whole process, try to recall:

- What do the "buckets" hold?
- Why we check items backwards?
- How do we determine the position of an item?
- Why sometimes we need to "kick-out" some elements, rather than simply add up the number of items in all buckets that are less than or equal to it?

The code of this algorithm could be rather confusing. In code, after computing how many items should each "bucket" holds, we replace them with how many items that are *less than or equal to* the "bucket label", that is, a *prefix sum* of the "buckets". If you redo the process above, you can discover that this will largely reduce the complexity of the process: when we determine the position of element x, we don't need to calculate how many items are less than or equal to x, we can directly read off from "bucket", also, when position of x is determined, we decrease the number stored in "bucket" x by 1, indicating *we have already ejected one x out, and we should not re-count it when we meet another x later.*

Algorithm 32: Counting-Sort(A, B, n, k)

Input: $A[1 \cdots n]$, where $A[i] \in \{1, 2, \cdots, k\}$
Output: $B[1 \cdots n]$, sorted A array
// $C[i]$ will be our "bucket" with label i
1 Let $C[1 \cdots k]$ be a new array with all 0
2 **for** $i \leftarrow 1$ *to* n **do**
  // For item $A[i]$, put it into "bucket" with label $A[i]$
3 $C[A[i]] \leftarrow C[A[i]] + 1$
4 **end**
5 **for** $j \leftarrow 2$ *to* k **do**
6 $C[j] \leftarrow C[j] + C[j-1]$ // prefix-sum, as we explained before
7 **end**
// Now $C[i]$ stores the number of items that $\leq i$
// Now check backwards, determine the position of each item
8 **for** $j \leftarrow n$ *to* 1 **do**
  // item $A[j]$ has $C[A[j]]$ items $\leq$ it, so it should at position $C[A[j]]$
9 $B[C[A[j]]] \leftarrow A[j]$
  // Remember we have ejected one $A[j]$, should not re-count when meet another $A[j]$ later
10 $C[A[j]] \leftarrow C[A[j]] - 1$
11 **end**

Running time: $\Theta(n + k)$.

This process is only slightly different from what we introduced above. I'd like to attach a picture in textbook for your reference, which shows the detail steps of what the algorithm given above is doing.

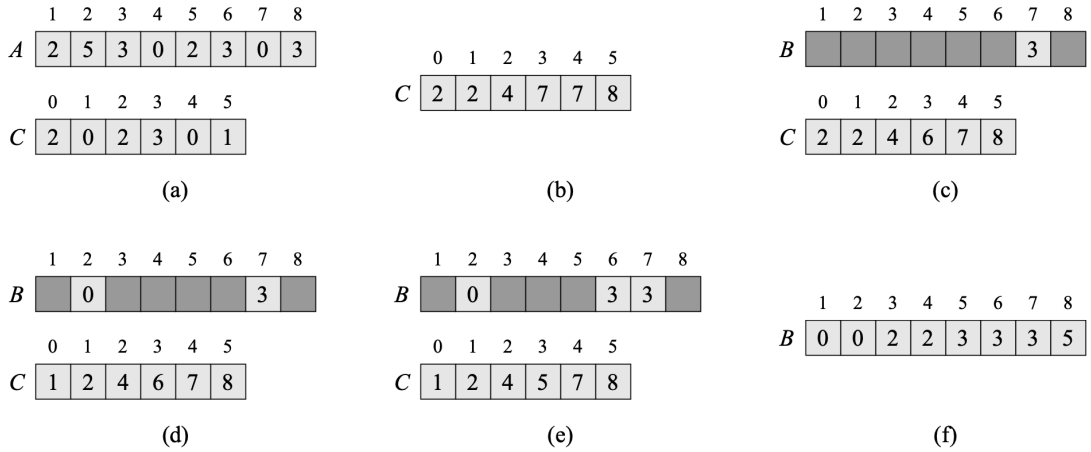


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

6.3 Radix Sort

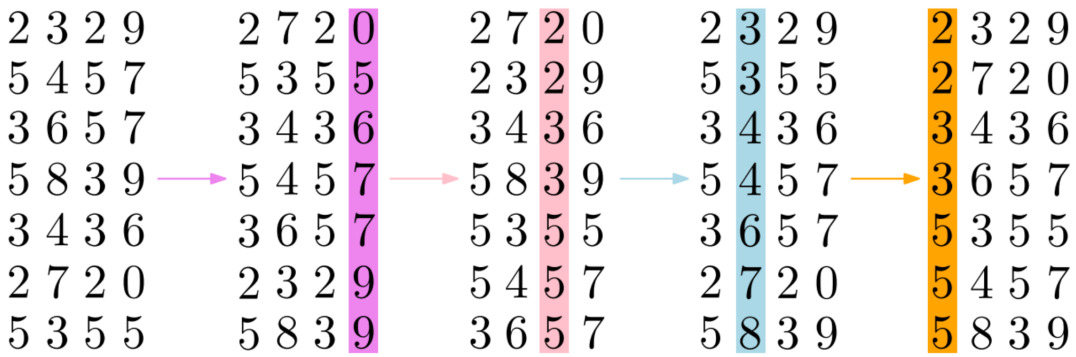
Radix Sort is quite interesting. With the help of a *stable sort*, where Counting Sort will be a good choice, it sorts the array on one digit, each time, from **least-significant bit**(LSB) to **most-significant bit**(MSB).

Here is what Radix Sort does, on an input of array of n items, each of them has d digits, and digit 1 is the right-most digit(LSB), digit d is the left-most digit(MSB).

Algorithm 33: Radix-Sort(A, d)

```
1 for i ← 1 to d do
2 |   use counting sort to sort array A on digit i
3 end
```

Here is an example of what Radix Sort does on each step:



Since here we are focusing on decimal numbers, so for counting sort, $k = 10$(each digit is in $\{0, 1, \cdots, k-1\}$). It is not necessary that we sort it on decimal numbers, actually each digit can be of any range, maybe hexadecimal, binary, etc, since all of them work fine with Counting Sort.

We prove the correctness of Radix Sort by induction.

Proof. Assume that the numbers are already sorted by their low-order $i-1$ digits. (that is, we've already use radix sort on digit $1 \cdots i-1$)

Now assume sorting on digit i.

- Two numbers that differ on digit i will be correctly sorted by their $i-$th digit,
- Two numbers that we same digit i will be put in the same order in output as they were in input, since we are using a stable sort, (Counting Sort, usually)

Therefore, they are correctly sorted by their $i-$th digits. ■

The running time of Radix Sort is depends on three variables: n, the size of input, k, the numbers of values that each digit can take, and d, number of digits in each item.

We know Counting Sort takes $\Theta(n+k)$ time, and for Radix Sort, we run Counting Sort for d times, its running time should be $\Theta(d(n+k))$.

If you remember what we said above, it is not necessary that we represent each item in decimal numbers. If we are given n hexadecimal numbers, k changes from 10 to 16, but d will become smaller. On the contrary, if we are given n binary numbers, k will be only 2, but d will become larger.

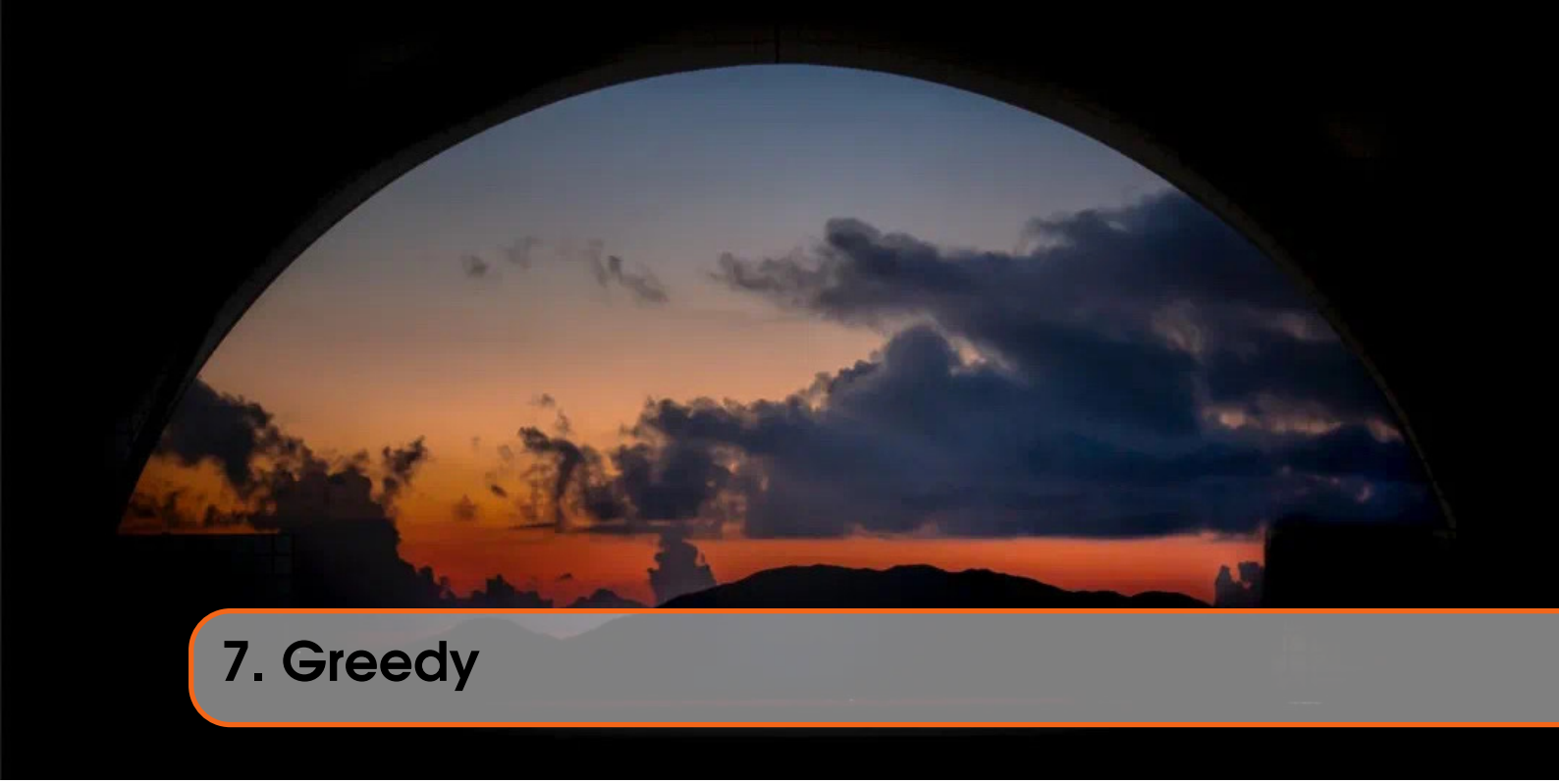
6.4 Sort Review

For now, you've seen those sorting algorithms.

	Insertion sort	Merge sort	Quicksort	Heapsort	Radix sort
Running time	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta\big(d(n+k)\big)$
Randomized	No	No	Yes	No	No
Working space	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n+\mathrm{k})$
Comparison-based	Yes	Yes	Yes	Yes	No
Stable	Yes	Yes	No	No	Yes

Please notice that though linear sort algorithms exist, we still consider the running time of sorting algorithms are generally $O(n\log n)$, since for linear sort algorithms, more requirements are required, such as the numbers must be integer, the number must lie in a certain (and not too large) range, etc.

III Advanced Design Techniques



7. Greedy

7.1 Introduction

Let's first define(informally) what is a **Greedy Algorithm**. Consider for a problem that requires us to find some kinds of *optimal solutions*, and to find it, we always makes the choice that *looks best at current moment*, and choose it as a part of our final solution.

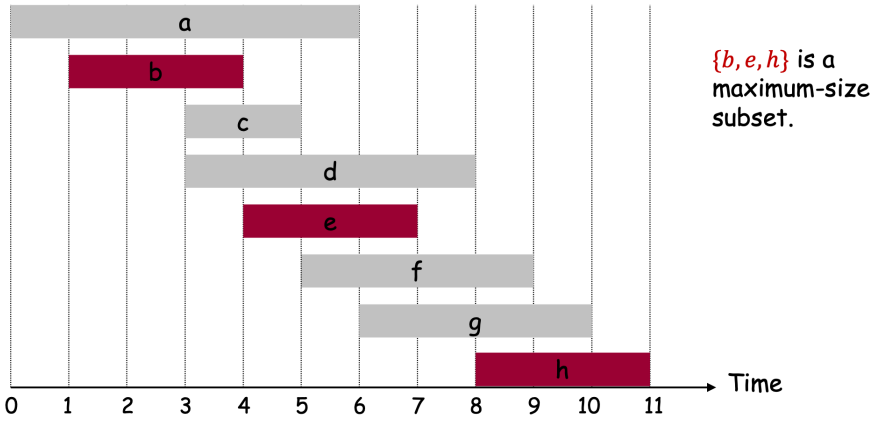
This idea may sound natural, simple, and even runs efficiently. Indeed it is, however, one may alert that this strategy doesn't always give us the optimal solution *of the whole problem*(we only make optimal choice at each step, we cannot guarantee the solution is still optimal after all steps). We will see some problems that greedy algorithm fails to find optimal.

As you can see, greedy algorithm is really easy, while the hardest part is how to formally *prove* it. You must convince other people that your greedy algorithm is able to find *global optimal solution* by simply grab *optimal solution at each step*. We will cover three different kinds of problems(except for Huffman Coding, which is a special one) where greedy algorithm performs well, and the first two have similar proof of correctness, while the third one has a totally different proof. (You may be required to redo the proof or adapt the proof to another similar problem in homework/exam)

7.2 Interval Scheduling

Problem: Given n jobs, where job j starts at time s_j and finishes at f_j. We say two jobs i and j are *compatible* or *not overlap* if their time intervals $(s_i, f_i) \cap (s_j, f_j) = \emptyset$. Our goal is to find

the maximum-size subset of *mutually compatible* jobs.



In the example above, $\{b, e, h\}$ is a maximum-size subset, since you cannot find a subset of size more than 4 and the jobs in that subset are mutually compatible.

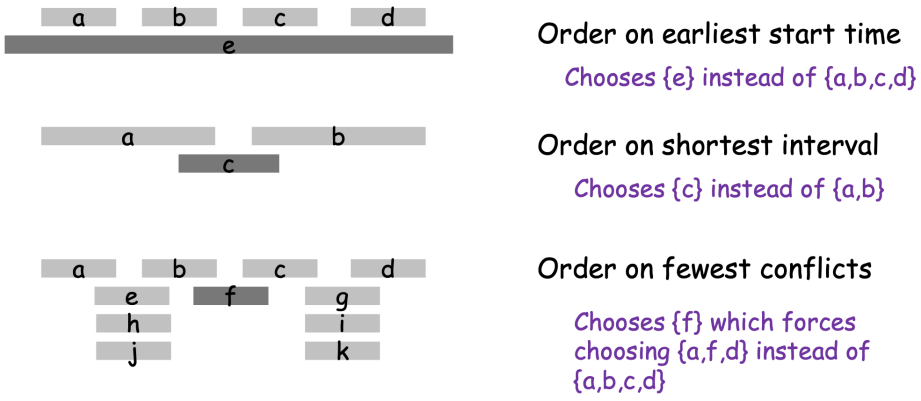
We consider to solve this problem by greedy, that is, given some jobs, we need to make the best choice at current timestamp, that could make the size of subset maximized. So how can we maximized it? Consider choosing a job if it is compatible with all previous ones that we've already taken, and not choosing it otherwise. In this way we achieve what we said that "make the best current choice".

However, you may easily find a counterexample, like if we first encounter a job that overlap with all other jobs, then we can choose at most one job using the strategy above. So to ensure its correctness, we need to first *consider jobs in some order*, and then use the above method.

This order can be hard to decide, and one may think of sort jobs by:

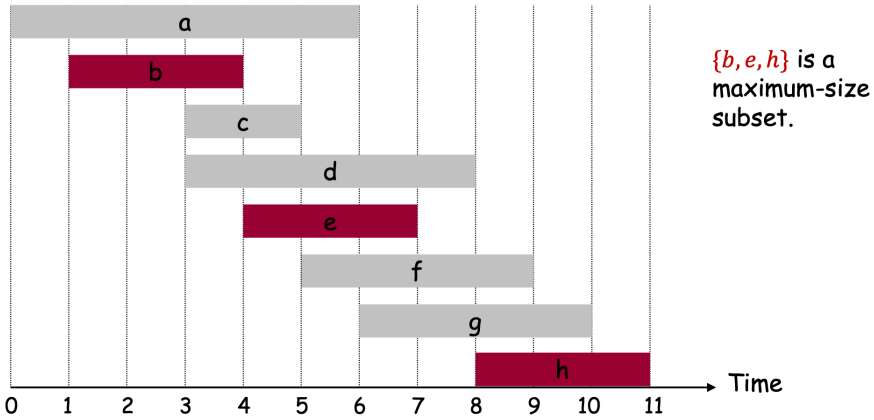
1. increasing order of *start time* s_j,
2. increasing order of *interval length* $f_j - s_j$,
3. increasing order of *fewest conflicts* of each job,
4. increasing order of *finish time* f_j.

Unfortunately, we can find(not difficult) counterexamples for first three orders:



The fourth order, i.e., sort jobs by *finish time*, can actually guarantee the greedy algorithm to be optimal. This is not easy to find out, but the intuition is that we can leave *maximum time* for scheduling *the rest of jobs*.

You may want to review this image:



We first give the code that implements this algorithm, and then formally prove it.

Algorithm 34: Interval-Schedule($(s_1, f_1), \cdots, (s_n, f_n)$)

```
1  Sort jobs by finish time so that f_1 ≤ f_2 ≤ ··· ≤ f_n
2  A ← ∅      // A is the set to store the subset we found
3  last ← 0      // last records the finish time of last job we've chosen
4  for j ← 1 to n do
       // If job j is compatiable with previous jobs, i.e., starts no earliear
          than the finish time of last job we've chosen
5      if s_j ≥ last then
6          A ← A ∪ {j}      // Add job j into our set
7          last ← f_j      // Update last
8      end
9  end
10 return A
```

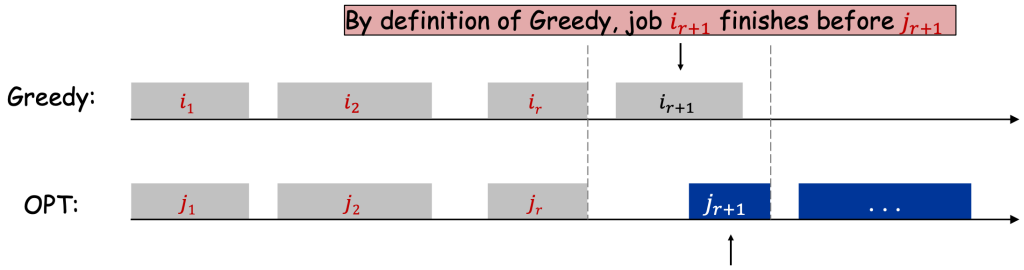
The running time of this algorithm is dominated by sorting process, which causes $\Theta(n \log n)$.

Now let's prove the correctness of this algorithm, i.e., *the greedy algorithm is optimal.*

Assume the set of jobs we get from greedy algorithm is different from an optimal set. Let $i_1, i_2, \cdots, i_k$ be the set of jobs found by greedy algorithm, while $j_1, j_2, \cdots, j_m$ be the set of jobs in optimal solution. Our basic idea is to *modify* optimal solution into the same as our greedy solution, while at the same time *ensure the solution is still optimal after modification.*

We start by finding largest possible value of r such that $i_1 = j_1, i_2 = j_2, \cdots, i_r = j_r$, that is, find the last job that are the same in both sets. Thus, $i_{r+1} \neq j_{r+1}$.

Notice in our greedy algorithm, we sort jobs by increasing of finish time, so if $i_{r+1} \neq j_{r+1}$, there must be job i_{r+1} finishes no later than j_{r+1}, i.e., $f_{i_{r+1}} \leq f_{j_{r+1}}$.



Then, we try to modify optimal solution, say, create $OPT^\star$ from OPT by just replacing j_{r+1} with i_{r+1}. Notice now $OPT^\star$ is still a legal solution, the jobs are still mutually compatible, and $OPT^\star$ has the same size as OPT, so $OPT^\star$ is also an optimal solution!

Let's do this process again, until the optimal solution is the same as greedy. A minor problem is that they may not have the same size, i.e., $k \neq m$. If this happens, the only possible is $k < m$, since m is the size of optimal solution, it must be the largest! However, $k < m$ is also impossible, since after replacing all jobs using above method, $i_1 = j_1, i_2 = j_2, \cdots, i_k = j_k$. If there is still jobs in optimal solution, say j_{k+1}, that is compatible with all previous jobs, then our greedy algorithm must would have chosen that(this holds immediately from our algorithm). So there must be $k = m$.

7.3 Knapsack

Problem: Given n items, where item i has weight w_i and value v_i. You now only has a knapsack that can holds weight at most W, you want to fill your knapsack with as much *value* as possible.

Notice in this problem an item can be *partially used*, like each item is some liquid.

The idea is quite simple: since we can partially use an item, we just first choose items with largest "value-weight ratio", i.e., largest $\frac{v_i}{w_i}$. If the bag is full, we stop choosing, otherwise, we repeat the process on remaining items.

Algorithm 35: Fractional-Knapsack$(w_1, v_1, w_2, v_2, \cdots, w_n, v_n, W)$

```
1  Sort items so that v_1/w_1 ≥ v_2/w_2 ≥ ··· ≥ v_n/w_n
2  w ← W     // w records how many more weight can the knapsack hold
3  for i ← 1 to n do
       // If w_i ≤ w, we can choose whole item, otherwise, we can only choose
          partial
4      if w_i ≤ w then
5          x_i ← 1     // x_i records how much of the item is chosen, 1 means whole
6          w ← w − w_i     // update remaining capacity of the knapsack
7      else
8          x_i ← w/w_i     // choose partial
9          return     // No need to update w, since we are already done
10     end
11 end
12 return
```

Again, its running time depends on the sorting process, which is $\Theta(n \log n)$.

To prove the correctness of this greedy algorithm, we can assume $\sum_{i=1}^{n} w_i \geq W$, i.e., the knapsack is fully packed. Otherwise, the algorithm just takes all items thus is trivially optimal.

For the following proof, we assume the items are already *sorted by* "value-weight ratio".

We will use the technique in last example, let the greedy solution be $G = (x_1, x_2, \cdots, x_k, 0, \cdots, 0)$, where for $i < k, x_i = 1$, since we fully take those items, and for k-th item, $0 \leq x_k \leq 1$, since we may or may not fully take it, and for $i > k, x_i = 0$, since they are not so "valuable" and we didn't take it. Also consider some optimal solution $OPT = (y_1, y_2, \cdots, y_n)$. Note that since both of them must fully pack the knapsack, there must be:

$$\sum_{i=1}^{k} x_i w_i = \sum_{i=1}^{n} y_i w_i = W$$

Again, we find the first item i where G and OPT differ:

$$\begin{array}{rcccccccccccc} G & = & x_1 & x_2 & x_3 & \cdots & x_{i-1} & x_i & \cdots & x_k & \cdots & 0 & 0 \\ OPT & = & x_1 & x_2 & x_3 & \cdots & x_{i-1} & y_i & \cdots & y_k & \cdots & y_{n-1} & y_n \end{array}$$

Since our algorithm is greedy, it will take item i as much as possible, so there must be $x_i > y_i$.

We let $\Delta x = x_i - y_i > 0$.

We can then modify the OPT as follows:

- change y_i to x_i, i.e., take as much as what we've taken in greedy
- remove some items in $i+1, \cdots, n$, such that their total weights equal to Δx.
- What we have done up to now is that we take more item i while throw away equal weights of $i+1, \cdots, n$ items.
- Since those items are sorted by "value-weight ratio", as stated before, the value of OPT after modification *must have not decreased.*(think why)
- Since the value of OPT must have already been the largest before modification, so the value of OPT must not change.
- So the new solution $OPT^\star$ must still be an optimal solution.

In the same way, we repeat this process, and will eventually convert OPT to G.

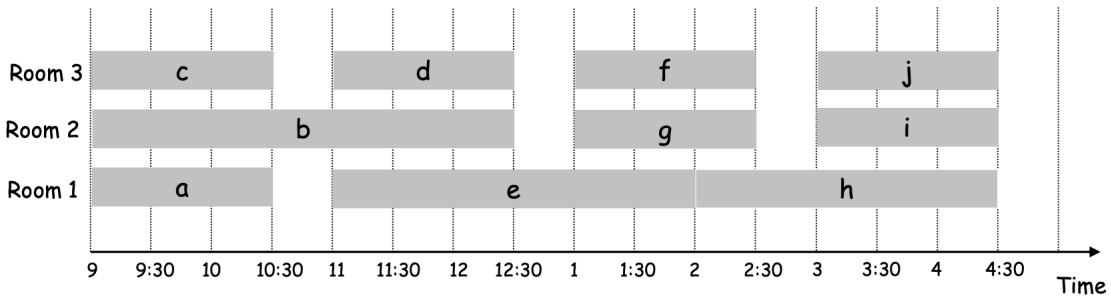
Hence our greedy algorithm yields an optimal solution, the correctness is proved.

One thing worths mentioning is that if this problem is modified into "items cannot be partially used", i.e., you can only either choose the item, or not choose the item(also known as "0/1 Knapsack Problem"), then there is no greedy algorithm that yields optimal solution. We will come back to this in Dynamic Programming.

7.4 Interval Partitioning

Problem: There are n lectures, where lecture j starts at time s_j and finishes at time f_j. Our goal is to find the *minimum number of classrooms* to schedule ALL lectures so that no two occurs at the same time in the same room.

For example, the below schedule, which uses 3 classrooms, is the optimal solution for given lectures.



For this problem, we sort lectures in increasing order of *start time*, and assign each lecture to any compatible classroom. That is, for each lecture j, if there is an existing classroom k such that k is empty during time interval (s_j, f_j), then we assign lecture j to classroom k.

Otherwise, if all existing classrooms are incompatible with the lecture, we allocate it to a new classroom.

The algorithm is shown below:

Algorithm 36: Interval-Partition($(s_1, f_1) \cdots (s_n, f_n)$)

```
1  Sort intervals by starting time so that s_1 ≤ s_2 ≤ ··· ≤ s_n
2  d ← 0   // d records # of classrooms used so far
3  for j ← 1 to n do
4  |   if lecture j is compatible with some classroom k then
5  |   |   schedule lecture j in classroom k
6  |   else
7  |   |   allocate a new classroom numbered d + 1
8  |   |   schedule lecture j in classroom d + 1
9  |   |   d ← d + 1
10 |   end
11 end
12 return d
```

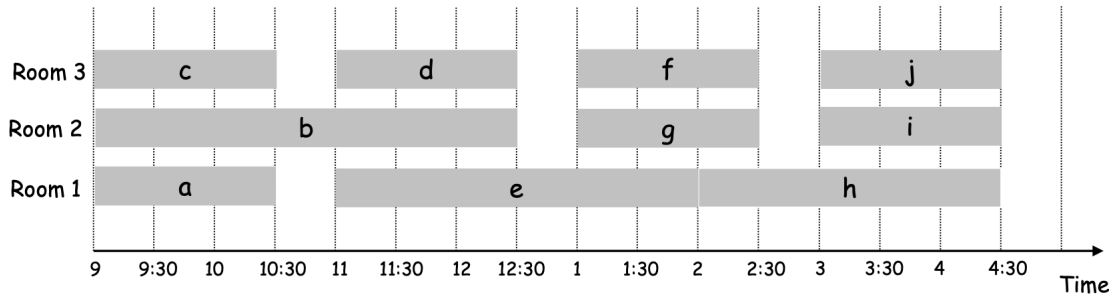
(Before we prove this is optimal, you should convince yourself that sorting by finish time does not yield optimal.)

Firstly, we define "depth" of a set of lectures(or generally, a set of open intervals).

Definition 7.4.1 The **depth** of a set of open intervals is the maximum number that exist at any instant of time.

Intuitively, at each second of the day, you record how many *simultaneous classes* are being taught at that second, and the "depth" is the max number of all those numbers you have recorded.

Unsurprisingly, the minimum number of classrooms required must be $\geq$ depth. For example, back to our example at the beginning, the **depth of the lectures** is 3, so we expect we should at least use 3 classrooms. Well, interestingly, the schedule we give below uses just 3 classrooms. Thus, we are sure that this schedule is optimal.



Now, to prove our greedy algorithm is optimal, we only need to show the # classrooms used by our greedy algorithm *equals to* "depth".

Theorem 7.4.1 The number of classrooms used by greedy, say d, is no more than the **depth** of the set of lectures, i.e.,

$$\text{depth} \geq d$$

Proof. Let d be the number of classrooms used by greedy.

- Classroom d is opened because we need to schedule a lecture j, while j is incompatible with all $d-1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that are *start no later than* s_j, and *finish later than* s_j.
- Thus, there $\exists \epsilon > 0$, where at time $s_j + \epsilon$, all d lectures(the $d-1$ incompatible ones and lecture j) are overlapping.
- Therefore, there are at least d lectures at time $s_j + \epsilon$
- $\Rightarrow$ depth $\geq d$

Since the number of classrooms used by greedy, d, is $\leq$ depth, our greedy is optimal. ■

Now back to our algorithm, let's analyze its running time. (see codes below)

To achieve line (*), we need to maintain for each classroom, the finishing time of the last lecture placed in that classroom. So at line (*), we compare s_j with those finishing times. If $s_j \geq$ any one of those finishing time, we can schedule lecture j into that classroom.

Therefore, line (*) uses $O(n)$ time to check, this algorithm runs in $O(n^2)$ time.

However, it is possible to improve it. Consider if the finishing time of classrooms are

$$[3, 5, 6, 4, 9],$$

and now we have a lecture j such that $s_j = 2$. Notice that since s_j is incompatible with the first classroom, which ends at 3, then we don't bother to check all remaining ones! 3 is the *earliest*

Algorithm 37: Interval-Partition($(s_1, f_1) \cdots (s_n, f_n)$)

```
1  Sort intervals by starting time so that s_1 ≤ s_2 ≤ ··· ≤ s_n
2  d ← 0  // d records # of classrooms used so far
3  for j ← 1 to n do
4  |  if lecture j is compatible with some classroom k      (*) then
5  |  |  schedule lecture j in classroom k      (**)
6  |  else
7  |  |  allocate a new classroom numbered d + 1
8  |  |  schedule lecture j in classroom d + 1      (***)
9  |  |  d ← d + 1
10 |  end
11 end
12 return d
```

finishing time among all classrooms, if j is not compatible with that one, it cannot be compatible with any other classrooms. On the contrary, if we have a lecture k such that $s_k = 3$, then we also only need to check the first classroom, which ends at 3, and they are compatible! And we still need to check others.

Hence, we could maintain the classrooms in a **min priority queue**, using the finishing time of the last class in that room as the key value. To implement (*), we do **extract-min** to get the *earliest finishing time* among all classrooms f_{min}, and check whether $f_{min} \leq s_j$. If $f_{min} > s_j$, we need a new classroom.

For line (**), we just need to add the new finishing time f_j to priority queue.

For line (***), we *re-insert* f_{min} back to priority queue, (since we extracted it before) AND insert the new finishing time f_j into priority queue.

Under this implementation, all (*), (**), and (***) lines are $O(\log n)$ jobs, so the algorithm runs in $O(n \log n)$ time.

7.5 Huffman Coding

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by table below. (i.e., there are only 6 different characters)

	a	b	c	d	e	f
Frequency(in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

To compact the file, we consider designing a **binary character code**, in which each character is represented by a unique binary string, which we call a **codeword**. If we use a **fixed-length code**, we need 3 bits to represent 6 characters, since $2^2 < 6 < 2^3$. This method requires

$$(45 + 13 + 12 + 16 + 9 + 5) \cdot 3 = 300 \text{ k bits}$$

to code the entire file. Can we do better?

Actually, instead of **fixed-length code**, we can use **variable-length code**. If we give frequent characters short codewords and infrequent characters long codewords, as the table above shows, this method only requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) = 224 \text{ k bits}$$

to code the entire file. Believe it or not, this is an optimal way to encode this file.

8. Dynamic Programming: 1D

8.1 Introduction to DP

Consider how to calculate the n-th term of Fibonacci Sequence, which is given by:

$$f_1 = 1,\ f_2 = 1,\ f_n = f_{n-1} + f_{n-2},\ \text{for } n \geq 3$$

Firstly, since we have been provided with the *recurrence*, we may think of using recursion to solve:

Algorithm 38: Fib-Recursion(n)

```
1 if n = 1 or n = 2 then
2 |   return 1
3 else
4 |   return Fib-Recursion(n − 1) + Fib-Recursion(n − 2)
5 end
```

The running time of this algorithm is given by

$$T(n) = T(n-1) + T(n-2)$$

observe that

$$T(n) \geq 2T(n-2) \geq 2^2 T(n-4) \geq 2^3 T(n-6) \geq \cdots \geq 2^{n/2},$$

while on the other side

$$T(n) \leq 2T(n-1) \leq 2^2 T(n-2) \leq \cdots \leq 2^n,$$

hence the running time is between $2^{n/2}$ and 2^n, this is not an efficient algorithm.

The reason why it runs so slowly is that we have solved the same sub-problem for so many times. For example, when solving f_n, we solved sub-problems f_{n-1} and f_{n-2}, but when solving f_{n-1}, we solved f_{n-2} again. As you should convince yourself, we have done lots of repeated work.

One way to solve this problem is that we "memorize" the results that we have computed, and not re-compute it again in the future. This is called "memoization", which will not be discussed in this course.

Algorithm 39: Fib-Mem$(n, result[1 \cdots n] = \{nil\})$

```
1 if result[n] ≠ nil then
      // which means we have computed before, so we directly return the value
2     return result[n]
3 else
      // otherwise, we need to compute it, and store it into the array
4     result[n] ← Fib-Mem(n − 1, result[1 ··· n]) + Fib-Mem(n − 2, result[1 ··· n])
5     return result[n]
6 end
```

For this algorithm, since we only compute each item once, and the running time for each item is $O(1)$(since we directly use $f_n = f_{n-1} + f_{n-2}$), the total running time is $O(n)$.

However, dynamic programming(DP) has a totally different idea. Notice the previous two algorithms make use of *recursion*, which means it is done "top-down", while on the contrary, DP is done "bottom-up".

Algorithm 40: Fib-DP(n)

```
1 Let A be a new array of size n
2 A[1] ← 1, A[2] ← 2
3 for i = 3 to n do
4     A[i] ← A[i − 1] + A[i − 2]
5 end
6 return A[n]
```

As you may compare, if we want to calculate $A[i]$ in DP, we must have already calculated all values that we need, i.e., in this case, $A[i-1]$ and $A[i-2]$. So we say it is "bottom-up" since we always compute a value *after all its required values have been computed.*

The running time of DP is also $O(n)$.

Now you have known what Dynamic Programming is, let's look at some examples. In this chapters, all problems are "1D" DP problems, which, actually means the array for storing the results has only one dimension.

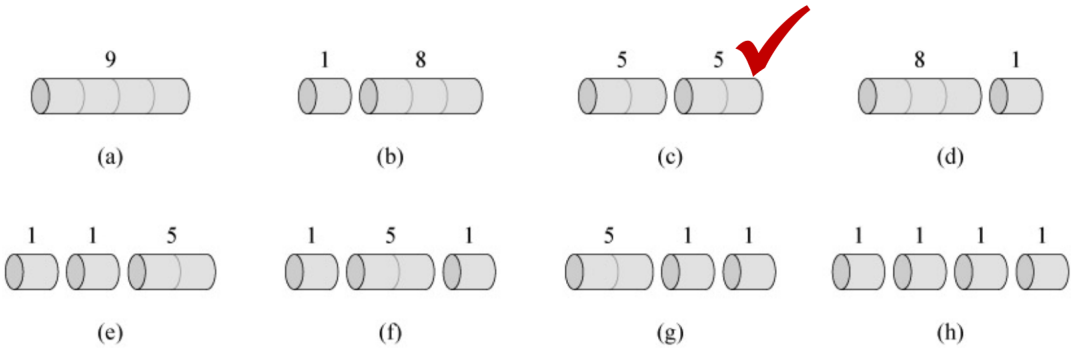
8.2 The Rod Cutting Problem

Problem: We are given the prices p_i of a rod of length i, where $i = 1, 2, \cdots, n$. Now there is a rod of length n, find a way to cut the rod so that we can maximize the total price.

For example, if the prices for different lengths of rods are given by

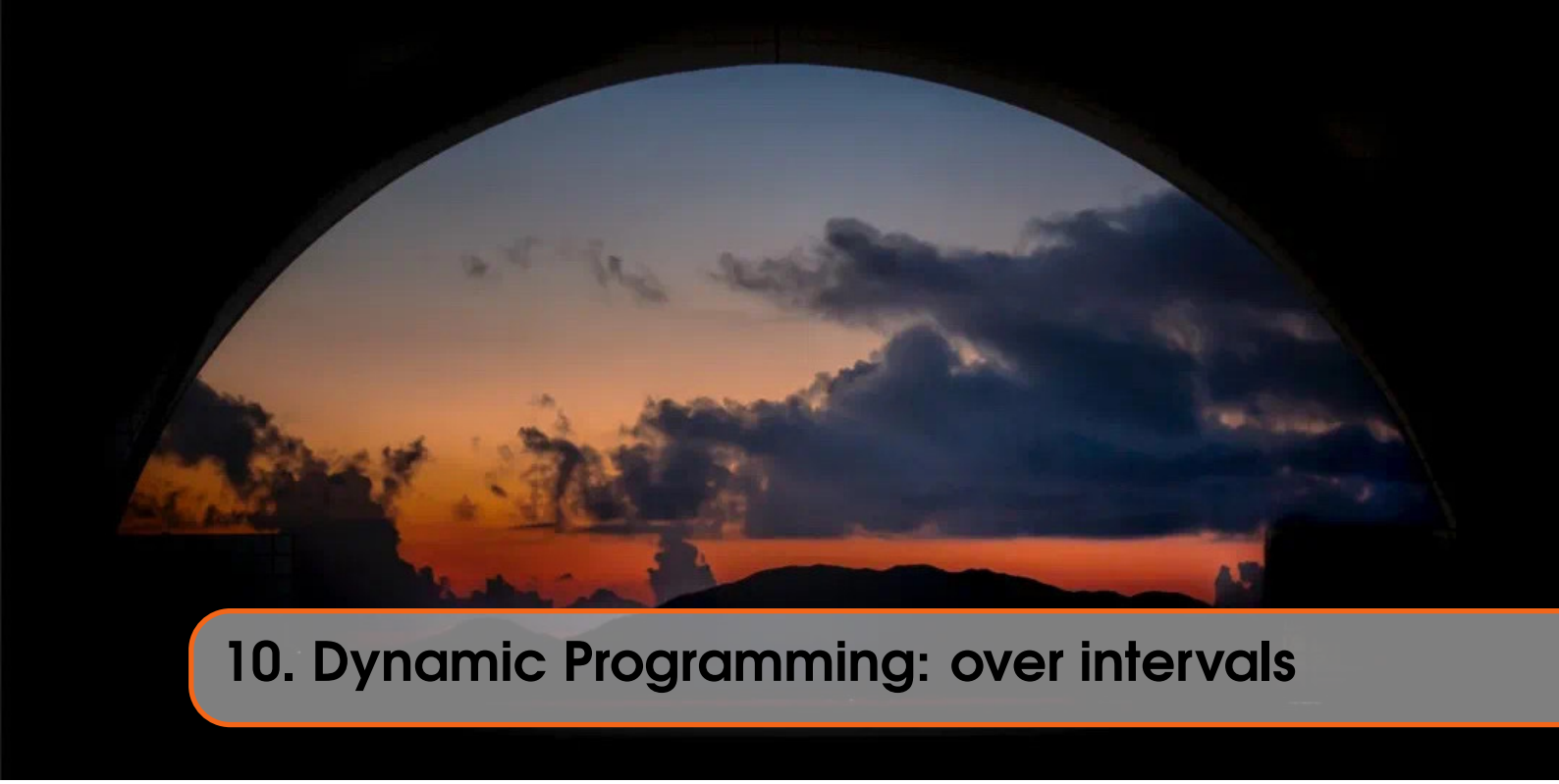
length i	1	2	3	4	5	6	7	8	9
price p_i	1	5	8	9	10	17	17	20	24

Then given a rod of length 4, among all possible cuttings, cutting it into two rods of length 2 has the max total price: $5 + 5 = 10$.



8.3 Weighted Interval Scheduling

9. Dynamic Programming: 2D



10. Dynamic Programming: over intervals