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.

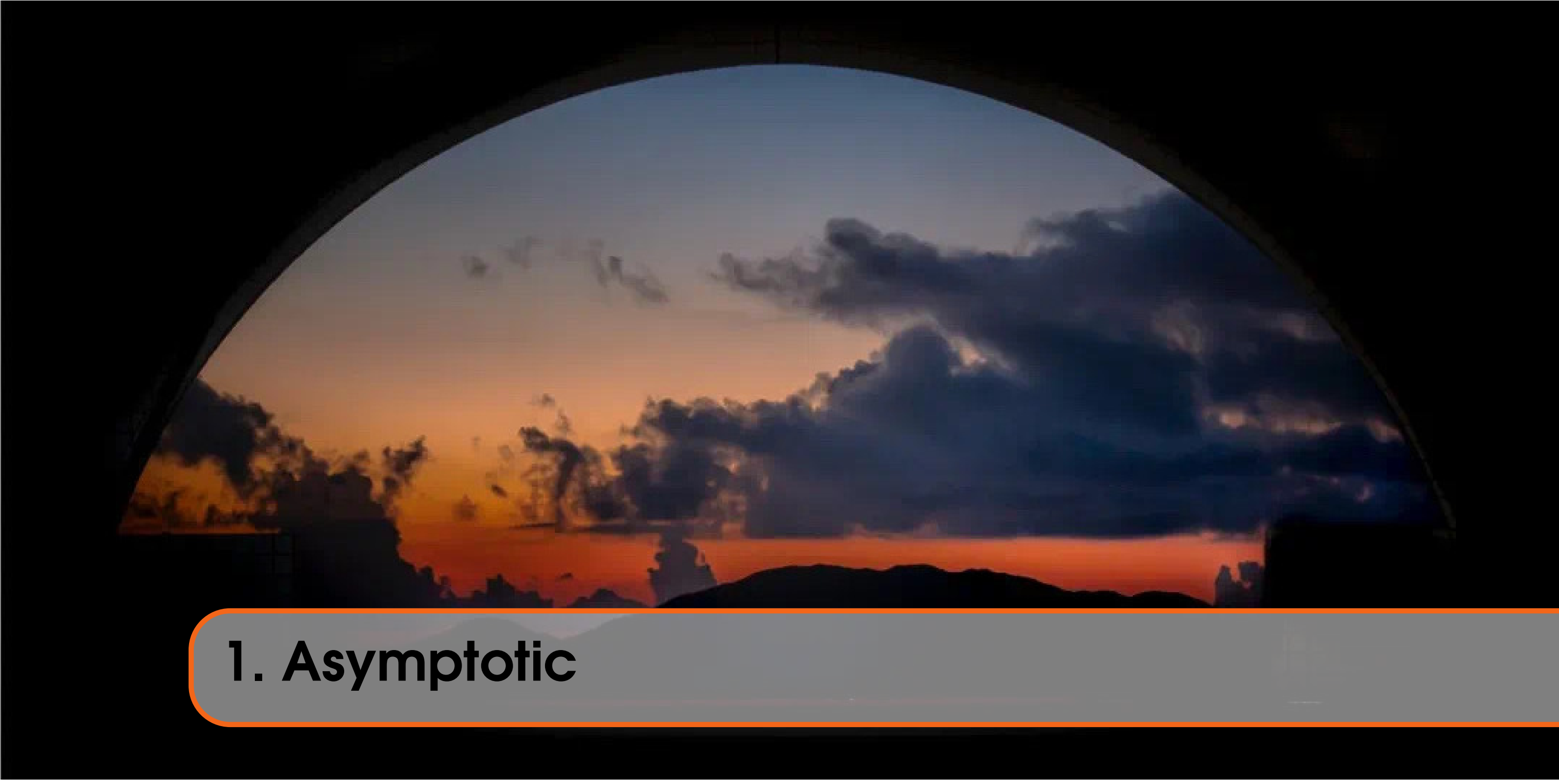
I have written this note in a very short space of time and as a result it may contain many errors and inaccuracies, readers are welcome to email me at `enor2017@163.com` or create an issue on my GitHub repo `https://github.com/enor2017/CourseNotes/issues`.

Hope you enjoy this book!

Contents

I Foundations

Foundations



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:

- **Upper bounds:** $T(n) = O(f(n))$, if $\exists c > 0, n_0 \geq 0$ such that $\forall n \geq n_0, T(n) \leq c \cdot f(n)$.
- **Lower bounds:** $T(n) = \Omega(f(n))$, if $f(n) = O(T(n))$.

- **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 $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} < \infty$, 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. $\displaystyle\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$, 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:

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 $n(n-1)/2$ 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: $E(I) = E(\sum X_{ij}) = \sum E(X_{ij})$.

That's a good one, $E(X_{ij})$ 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, $E(I) = \sum(1/2)$, and there are $n(n-1)/2$ terms, which gives $E(I) = n(n-1)/4 = \Theta(n^2)$.

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 1: 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
    $mid = \lfloor (left + right)/2 \rfloor$
    if $x \leq a[mid]$ 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

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

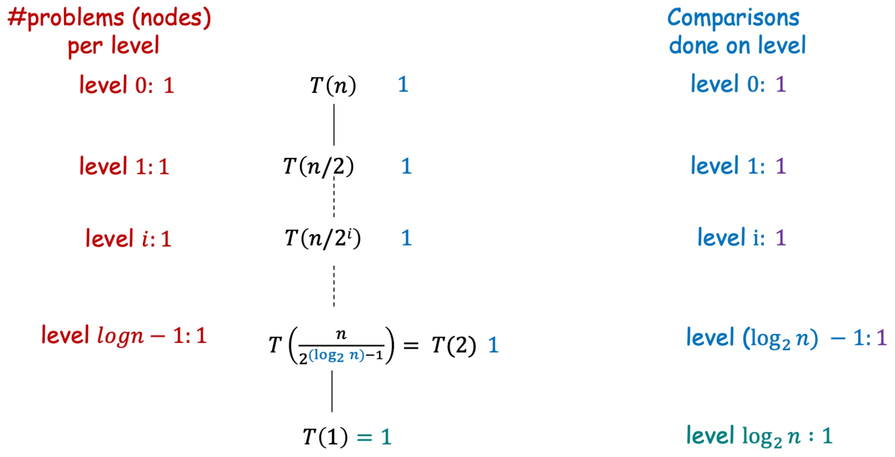
.

Solve this **recurrence**:

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

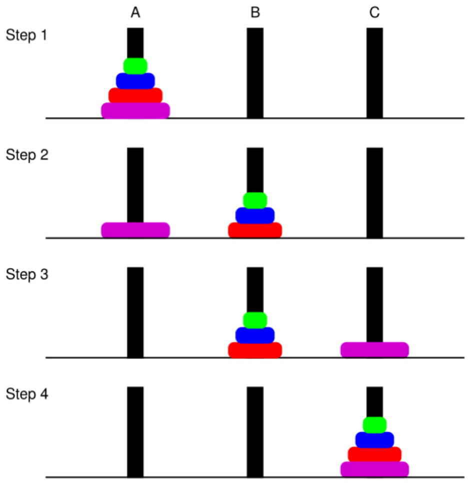
This process ends when reaching $T(1)$, i.e., $i = \log_2 n$, thus, $T(n) = T(1) + \log_2 n = \log_2 n + 1$.

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 2: MoveTower($n$, start, helper, end)
  Input: $n$: num of discs
  if $n = 1$ then
      move the only disc from start peg to end peg
      return
  else
      // move first $n-1$ from start peg to helper peg
      // so this time "helper" peg will be the old end peg
      MoveTower($n-1$, start, end, helper)
      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
      MoveTower($n-1$, helper, start, end)
  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:

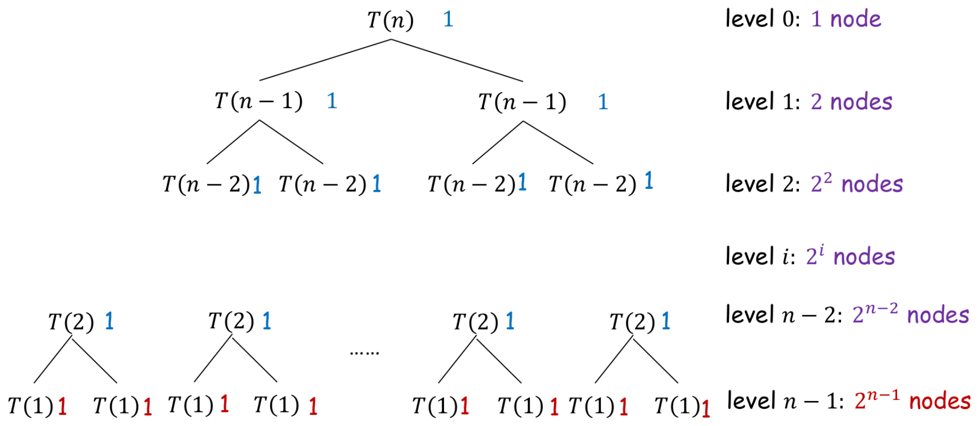
$$T(n) = 2T(n-1) + 1$$

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:

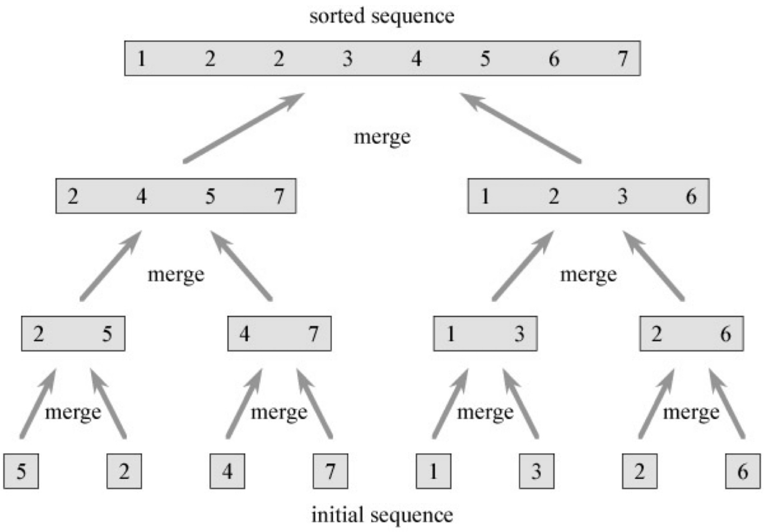
(1) Compare first items: $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $2 < 3$, so $result = [2]$;
(2) then compare 2nd in A and 1st in B, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $3 < 5$, so $result = [2, 3]$;
(3) continue the process, similarly, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $4 < 5$, so $result = [2, 3, 4]$;
(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 3: Merge(A, $left$, mid, $right$)

```
// merge two sorted list: $A[left \cdots mid]$ and $A[mid+1 \cdots right]$
$L \leftarrow A[left \cdots mid]$, $R \leftarrow A[mid+1 \cdots right]$
append $\infty$ at the end of $L$ and $R$      // see explanation above
$i \leftarrow 1$, $j \leftarrow 1$     // two pointers point at items in $L$ and $R$
for $k \leftarrow left$ to $right$ do
    // always choose the smaller one to insert, and move on
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i+1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j+1$
    end
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 4: MergeSort(A, $left$, $right$)

```
if $left = right$ then
    return
end
$mid \leftarrow \lfloor (left + right)/2 \rfloor$
// recursively divide array into two halves
MergeSort($A$, $left$, $mid$)
MergeSort($A$, $mid + 1$, $right$)
// then merge from bottom to up
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 $\frac{n}{2^k} = 1$ 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 5: Count(A, l, mid, r)

```
// $l$ means left, while $r$ means right.
// notice here, $L[1 \cdots (mid - l + 1)]$ is corresponding to $A[l \cdots mid]$, the
   subscript changes, try not be confused later.  $R$ also changes.
$L \leftarrow A[l \cdots mid]$, $R \leftarrow A[mid + 1 \cdots r]$
(here assume) $L, R$ already sorted
$i \leftarrow 1$, $j \leftarrow 1$    // two pointers for $L$ and $R$
$ans \leftarrow 0$    // total inversion number
// let $i, j$ iterator over two arrays
while $i \leq mid - l + 1$ and $j \leq r - mid$ do
    // looking for smallest $L$ item larger than $R$
    if $L[i] \leq R[j]$ then
        $i \leftarrow i + 1$
    else
        // Found $L[i] > R[j]$!
        // then $L[i] \cdots L[mid - l + 1]$ 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 $mid - l + 1 - i + 1 = mid - l - i + 2$
        $inv \leftarrow (mid - l - i + 2)$
        $ans \leftarrow ans + inv$
        $j \leftarrow j + 1$
    end
end
return $ans$
```

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

Algorithm 6: Count-Inversion(A, l, r)

```
if $l = r$ then
    return 0
end
$mid \leftarrow \lfloor (l + r)/2 \rfloor$
$c_1 \leftarrow$ Count-Inversion($A$, $l$, $mid$)
$c_2 \leftarrow$ Count-Inversion($A$, $mid + 1$, $r$)
MergeSort($A$, $l$, $mid$)
MergeSort($A$, $mid + 1$, $r$)
$c_3 \leftarrow$ Count($A$, $l$, $mid$, $r$)
return $c_1 + c_2 + c_3$
```

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

```
// same as previous algorithm, subscripts for $L, R$ and $A$ are different,
   remember this
$L \leftarrow A[l \cdots mid]$, $R \leftarrow A[mid + 1 \cdots r]$
append $\infty$ at the end of $L$ and $R$
$i \leftarrow 1$, $j \leftarrow 1$      // two iteration pointers for $L$ and $R$
$count \leftarrow 0$      // counter for inversion number
for $k \leftarrow l$ to $r$ do
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
        $count \leftarrow count + (mid - l - i + 2)$
    end
end
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 8: Sort-and-Count(A, l, r)

```
if $l = r$ then
    return 0
end
$mid \leftarrow \lfloor (l + r)/2 \rfloor$
$c_1 \leftarrow$ Sort-and-Count($A$, $l$, $mid$)
$c_2 \leftarrow$ Sort-and-Count($A$, $mid + 1$, $r$)
$c_3 \leftarrow$ Merge-and-Count($A$, $l$, $mid$, $r$)
return $c_1 + c_2 + c_3$
```

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 9: Max-Subarray-Brute-Force(A)

```
$maxSum \leftarrow A[1]$      // can also use -inf to initialize
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow i$ to $n$ do
        // calculate $A[i] + \cdots + A[j]$
        $sum \leftarrow 0$
        for $k \leftarrow i$ to $j$ do
            $sum \leftarrow sum + A[k]$
        end
        // if current $sum$ is larger, update $maxSum$
        if $sum > maxSum$ then
            $maxSum \leftarrow sum$
        end
    end
end
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 10: Get-Prefix-Sum(A)

```
// $S[]$ records the prefix sum of array $A$
$S[0] = 0$
for $i = 1$ to $n$ do
    $S[i] \leftarrow S[i-1] + A[i]$
end
return $S$
```

Algorithm 11: Max-Subarray-Prefix-Sum(A)

```
$maxSum \leftarrow A[1]$
$S \leftarrow$ Get-Prefix-Sum($A$)        // get prefix sum
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow i$ to $n$ do
        $sum \leftarrow S[j] - S[i-1]$        // calculate $A[i] + \cdots + A[j]$
        if $sum > maxSum$ then
            $maxSum \leftarrow sum$
        end
    end
end
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 12: Max-Subarray-Divide-Conquer(A, l, r)

```
if $l = r$ then
    return $A[l]$
end
$mid \leftarrow \lfloor (l+r)/2 \rfloor$
$max_1 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $l$, $mid$)
$max_2 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $mid+1$, $r$)
$max_3 \leftarrow$ Max Subarray that crosses the cut
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 13: Max-Subarray-Divide-Conquer(A, l, r)

```
if $l = r$ then
    return $A[l]$
end
$mid \leftarrow \lfloor (l+r)/2 \rfloor$
$max_1 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $l$, $mid$)
$max_2 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $mid+1$, $r$)
// now let's count $max_3$
$L_m \leftarrow -\infty$, $R_m \leftarrow -\infty$
$sum \leftarrow 0$
for $i = mid$ down to $l$ do
    $sum \leftarrow sum + A[i]$
    if $sum > L_m$ then
        $L_m \leftarrow sum$
    end
end
$sum \leftarrow 0$
for $i = mid + 1$ to $r$ do
    $sum \leftarrow sum + A[i]$
    if $sum > R_m$ then
        $R_m \leftarrow sum$
    end
end
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 14: 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 $S[j]$
$maxSum \leftarrow -\infty$, $minS \leftarrow 0$
// for each $j$, find $minS$, and then find $S[j] - minS$
for $j \leftarrow 1$ to $n$ do
    // update overall answer
    if $S[j] - minS > maxSum$ then
        $maxSum \leftarrow S[j] - minS$
    end
    // update $minS$ so far
    if $S[j] < minS$ then
        $minS \leftarrow S[j]$
    end
end
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 15: Max-Subarray-Linear2(A)

```
// $S$ will be all prefix sum so far, i.e., $A[1] + \cdots + A[j]$
$maxSum \leftarrow -\infty,\ minS \leftarrow 0,\ S \leftarrow 0$
for $j \leftarrow 1$ to $n$ do
    $S \leftarrow S + A[j]$        // calculate prefix sum so far
    if $S - minS > maxSum$ then
        $maxSum \leftarrow S - minS$
    end
    // update $minS$ so far
    if $S < minS$ then
        $minS \leftarrow S[j]$
    end
end
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 16: Max-Subarray-DP(A)

```
$d[0] \leftarrow 0$      // max sum of subarrays end with no item is 0
$maxSum \leftarrow A[1]$    // used to record max sum so far, notice here cannot
    initialize to 0
for $i = 1$ to $n$ do
    if $d[i-1] \leq 0$ then
        $d[i] \leftarrow A[i]$
    else
        $d[i] \leftarrow d[i-1] + A[i]$
    end
    if $d[i] > maxSum$ then
        $maxSum = d[i]$
    end
end
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 17: Max-Subarray-DP2(A)

```
$previousD \leftarrow 0$
$maxSum \leftarrow A[1]$
for $i = 1$ to $n$ do
    if $previousD \leq 0$ then
        $previousD \leftarrow A[i]$
    else
        $previousD \leftarrow previousD + A[i]$
    end
    if $previousD > maxSum$ then
        $maxSum = previousD$
    end
end
return $maxSum$
```

2.6 The Master Theorem

2.6.1 Theorem and its proof

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. 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 $T(n) = n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = \left(n^{\log_n a}\right)^{\log_b n} = n^{\left(\frac{\ln a}{\ln n} \cdot \frac{\ln n}{\ln b}\right)} = n^{\log_b a}$. ■

2.6.2 equalities, inequalities and more

to be added. (2021/09/05)

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 18: multiply-DC(A, B)

```
// $A[1\cdots n]$ and $B[1\cdots n]$ are two arrays storing string of base 10.
   $A[1], B[1]$ are least siginificant bits.(LSB)
$n \leftarrow$ size of $A$ and $B$
if $n = 1$ then
    return $A[1] \cdot B[1]$
end
$mid \leftarrow \lfloor n/2 \rfloor$
$M_1 \leftarrow$multiply-DC($A[mid+1\cdots n]$, $B[mid+1\cdots n]$)    // $x_h \times y_h$
$M_2 \leftarrow$multiply-DC($A[1\cdots mid]$, $B[mid+1\cdots n]$)    // $x_l \times y_h$
$M_3 \leftarrow$multiply-DC($A[mid+1\cdots n]$, $B[1\cdots mid]$)    // $x_h \times y_l$
$M_4 \leftarrow$multiply-DC($A[1\cdots mid]$, $B[1\cdots mid]$)    // $x_l \times y_l$
// 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.
$res[1\cdots n] \leftarrow M_4$
$res[mid+1\cdots] \leftarrow res[mid+1\cdots] + M_2 + M_3$
$res[n+1\cdots] \leftarrow res[n+1\cdots] + M_1$
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 19: Karatsuba(A, B)

```
// $A[1 \cdots n]$ and $B[1 \cdots n]$ 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,

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) & C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) & C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

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:

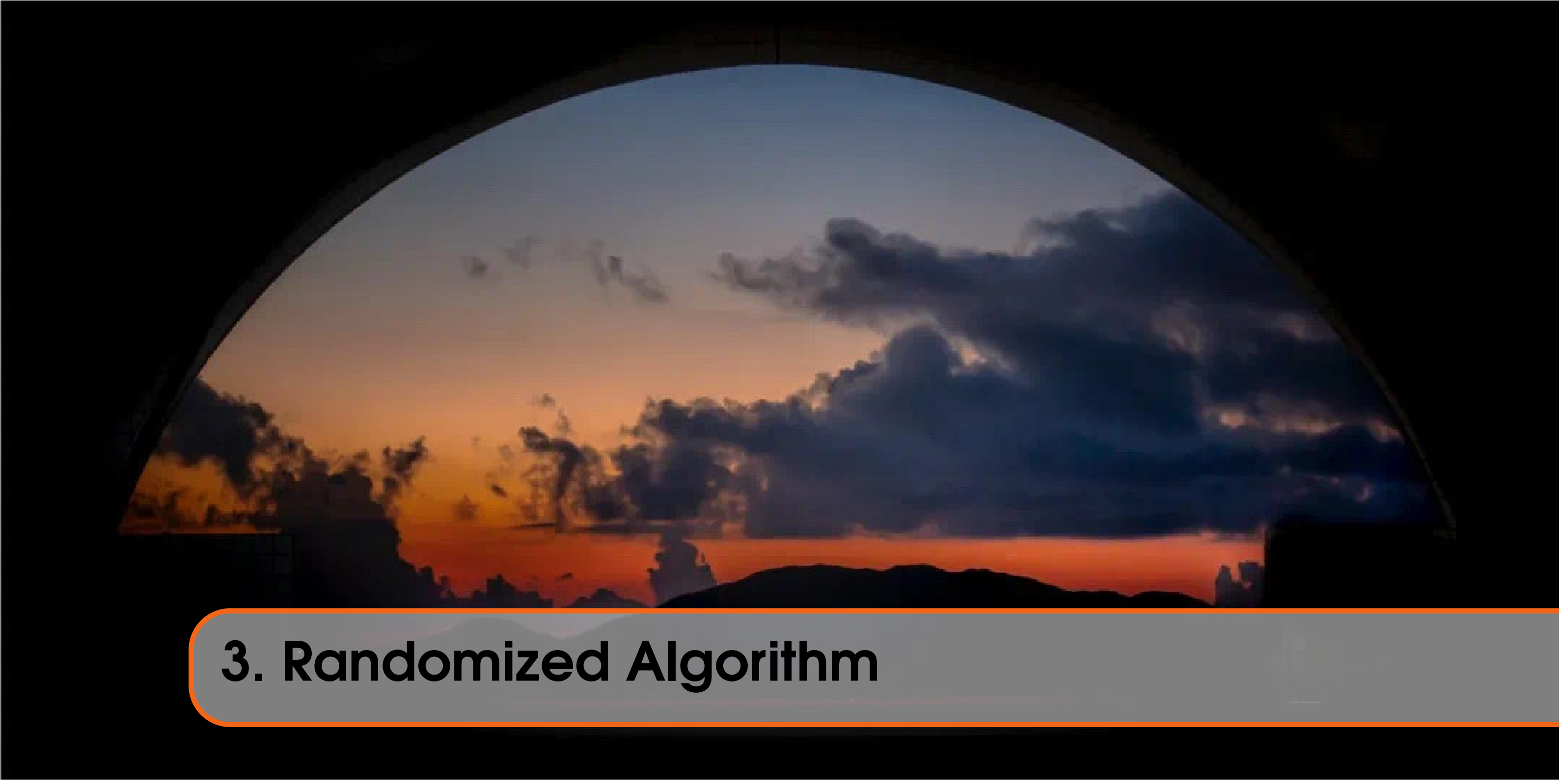
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

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

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

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 20: Hire-Assistant(n)

```
best ← 0
for i ← 1 to n do
    interview candidate i
    if candidate i is better than best then
        fire best
        hire candidate i
        best ← i
    end
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 21: Hire-Assistant($n$)
randomly permute all $n$ candidates
$best \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    interview candidate $i$
    if candidate $i$ is better than $best$ then
        fire $best$
        hire candidate $i$
        $best \leftarrow i$
    end
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

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

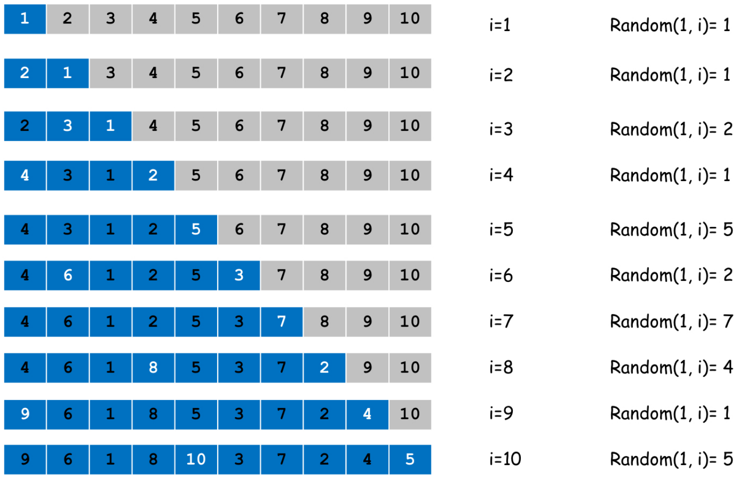
3.3 Generating a random permutation

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 22: RandomPermute($A$)
$n \leftarrow A.length$
for $i \leftarrow 1$ to $n$ do
    swap $A[i]$ with $A[Random(1, i)]$
end
```

Random Permutation: Example



1	2	3	4	5	6	7	8	9	10	i=1	Random(1, i)= 1
2	1	3	4	5	6	7	8	9	10	i=2	Random(1, i)= 1
2	3	1	4	5	6	7	8	9	10	i=3	Random(1, i)= 2
4	3	1	2	5	6	7	8	9	10	i=4	Random(1, i)= 1
4	3	1	2	5	6	7	8	9	10	i=5	Random(1, i)= 5
4	6	1	2	5	3	7	8	9	10	i=6	Random(1, i)= 2
4	6	1	2	5	3	7	8	9	10	i=7	Random(1, i)= 7
4	6	1	8	5	3	7	2	9	10	i=8	Random(1, i)= 4
9	6	1	8	5	3	7	2	4	10	i=9	Random(1, i)= 1
9	6	1	8	10	3	7	2	4	5	i=10	Random(1, i)= 5

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 23: QuickSort(A, l, r)

```
if $l \geq r$ then
    return
end
$pivot \leftarrow$ Partition($A$, $l$, $r$)
QuickSort($A$, $l$, $pivot - 1$)    // quick sort left part
QuickSort($A$, $pivot + 1$, $r$)    // quick sort right part
```

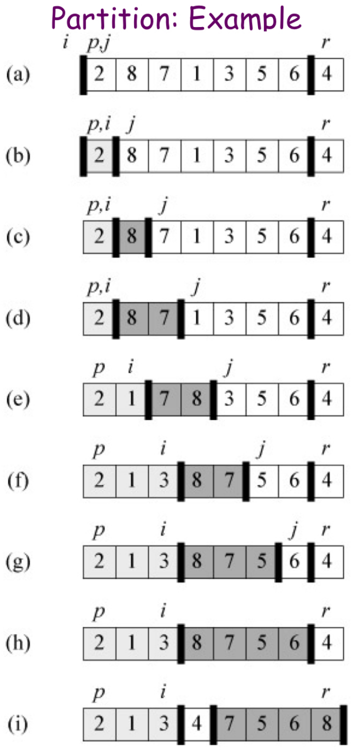
For the **Partition** process, we need to choose a pivot first, here we simply choose the last element as pivot. Then we divide the array $A[l \cdots r]$ into two parts: $A[l \cdots i]$ are all smaller or equal than pivot $A[r]$, $A[i+1 \cdots j-1]$ are all larger than pivot $A[r]$, while $A[j \cdots r-1]$ 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 24: Partition(A, l, r)

```
$x \leftarrow A[r]$      // choose last item to be pivot
$i \leftarrow l - 1$     // No item found yet, so there should be nothing in $A[l \cdots i]$
for $j \leftarrow l$ to $r - 1$ do
    if $A[j] \leq x$ then
        $i \leftarrow i + 1$      // expand left part
        swap $A[i]$ and $A[j]$      // include the new item into left part
    end
end
swap $A[i + 1]$ and $A[r]$      // make pivot to be at middle
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.

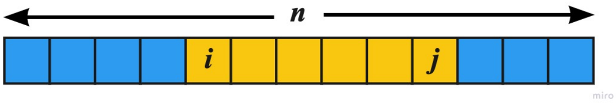
Without loss of generality, we can assume all elements are distinct, since if this is not the case, we can regard original input $A[1\cdots n]$ as $B[1\cdots n]$, where $B[i] = \{A[i], i\}$. Then all elements in B are distinct and we can just manipulate on $B[]$ instead. Also, remember the running time 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:

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

Then,

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

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

$\ldots$

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:

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

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.