CS3230 Semester 2 2024/2025

Design and Analysis of Algorithms

Tutorial 05
D&C, Sorting and Average-case analysis
For Week 06

Document is last modified on: February 4, 2025

MODEL ANSWER IS FOR OUR CLASS ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

1 Lecture Review: Decision Tree

A decision tree contains:

- Vertices (Internal): A comparison
- Branches: Outcome of comparison
- Leaves: Output / decision for the input

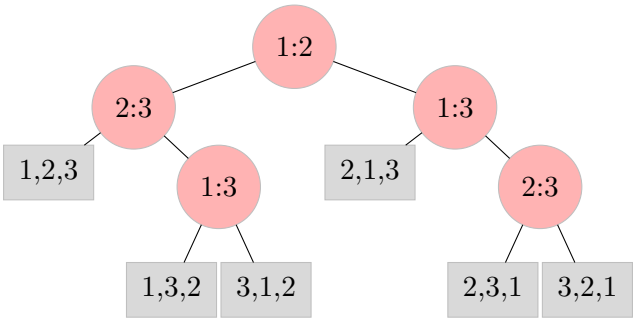

Figure 1: Worst case runtime is the height of the decision tree.

The classic example to illustrate the usage of decision tree is for showing the lower bound of comparison-based sorting is $\Omega(n \log n)$. Here is a picture of decision tree of sorting $n = 3$ elements and there are

$3! = 6$ possible outputs (decision for the inputs) which must all been catered for. As each comparison of two comparable elements a versus b yields two possible outcomes: $a < b$ (which means a must be in front of b) or $a \geq b$ (here b must be in front of a if $a > b$; on the other hand, if $a = b$, then it does not matter which order a and b are put in). This decision tree is thus a binary tree. The height of binary decision tree so that its number of leaves is at least $n!$ is $\log n! \approx n \log n$ (use Stirling's formula).

2 Tutorial 05 Questions

Q1, Q2, Q3, and Q4 involve Polynomial Multiplication of two polynomials of degree n.
Let $A(x) = a_n \cdot x^n + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$.
Let $B(x) = b_n \cdot x^n + \ldots + b_2 \cdot x^2 + b_1 \cdot x + b_0$.
Let $C(x) = A(x) \times B(x) = c_{2n} \cdot x^{2n} + \ldots + c_2 \cdot x^2 + c_1 \cdot x + c_0$
Assume all coefficients a_i, b_i, c_i are Integers.
Assume that all addition and multiplication operations of two Integers take $O(1)$ time.

We can compute the coefficients c_i of $C(x)$ in $O(n^2)$ using complete search:
For each $i \in [2n..0]$, $c_i = \sum a_j \cdot b_{i-j}$ where both j and $i - j$ are between 0 and n (inclusive).

Q1). Let $x = 10$ to make it easier to visualize this as a normal base 10 multiplication and $n = 2$.
Let $A(10) = 352 = 3 \cdot 10^2 + 5 \cdot 10 + 2$, i.e., $a_2 = 3, a_1 = 5, a_0 = 2$.
Let $B(10) = 221 = 2 \cdot 10^2 + 2 \cdot 10 + 1$, i.e., $b_2 = 2, b_1 = 2, b_0 = 1$.
Compute the coefficients of $C(10) = A(10) \times B(10) = 77\,792$ using the $O(n^2)$ algorithm above.

$c_4 = a_2 \cdot b_{4-2} = a_2 \cdot b_2 = 3 \cdot 2 = 6$.
$c_3 = a_1 \cdot b_{3-1} + a_2 \cdot b_{3-2} = a_1 \cdot b_2 + a_2 \cdot b_1 = 5 \cdot 2 + 3 \cdot 2 = 10 + 6 = 16$.
$c_2 = a_0 \cdot b_{2-0} + a_1 \cdot b_{2-1} + a_2 \cdot b_{2-2} = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 = 2 \cdot 2 + 5 \cdot 2 + 3 \cdot 1 = 4 + 10 + 3 = 17$.
$c_1 = a_0 \cdot b_{1-0} + a_1 \cdot b_{1-1} = a_0 \cdot b_1 + a_1 \cdot b_0 = 2 \cdot 2 + 5 \cdot 1 = 4 + 5 = 9$.
$c_0 = a_0 \cdot b_{0-0} = a_0 \cdot b_0 = 2 \cdot 1 = 2$.

In other words, $C(10) = 6 \cdot 10^4 + 16 \cdot 10^3 + 17 \cdot 10^2 + 9 \cdot 10 + 2 = 60\,000 + 16\,000 + 1700 + 90 + 2 = 77\,792$.

Q2). Suppose that you are given the following Divide and Conquer (D&C) algorithm:
Rewrite $A(x) = x^{\frac{n}{2}} \cdot A_1(x) + A_2(x)$
Rewrite $B(x) = x^{\frac{n}{2}} \cdot B_1(x) + B_2(x)$
where $A_1(x), A_2(x), B_1(x), B_2(x)$ are all now polynomials of degree (up to) $\frac{n}{2}$.

We now compute four smaller polynomial multiplications:
$A_1(x) \times B_1(x)$, $A_1(x) \times B_2(x)$, $A_2(x) \times B_1(x)$, $A_2(x) \times B_2(x)$

And we compute:
$C(x) = x^n \cdot [A_1(x) \times B_1(x)] + x^{\frac{n}{2}} \cdot [A_1(x) \times B_2(x) + A_2(x) \times B_1(x)] + A_2(x) \times B_2(x)$

Apply this D&C algorithm to compute the multiplication of the same two polynomials of degree $n = 2$:
Rewrite $A(10) = 352 = 10 \cdot (3 \cdot 10 + 5) + 2$
Rewrite $B(10) = 221 = 10 \cdot (2 \cdot 10 + 2) + 1$

Compute $A_1(10) \times B_1(10)$, $A_1(10) \times B_2(10)$, $A_2(10) \times B_1(10)$, $A_2(10) \times B_2(10)$.
Then, compute $C(10)$.

$A_1(10) \times B_1(10) = (3 \cdot 10 + 5) \times (2 \cdot 10 + 2) = 6 \cdot 10^2 + 16 \cdot 10 + 10 = 600 + 160 + 10 = 770$
$A_1(10) \times B_2(10) = (3 \cdot 10 + 5) \times 1 = 3 \cdot 10 + 5 = 35$
$A_2(10) \times B_1(10) = 2 \times (2 \cdot 10 + 2) = 4 \cdot 10 + 4 = 44$
$A_2(10) \times B_2(10) = 2 \times 1 = 2$

$$\begin{aligned} C(10) &= 10^2 \cdot (A_1(10) \times B_1(10)) + 10 \cdot (A_1(10) \times B_2(10) + A_2(10) \times B_1(10)) + A_2(10) \times B_2(10) \\ &= 10^2 \cdot \left(6 \cdot 10^2 + 16 \cdot 10 + 10\right) + 10 \cdot (3 \cdot 10 + 5 + 4 \cdot 10 + 4) + 2 \\ &= 6 \cdot 10^4 + 16 \cdot 10^3 + 10 \cdot 10^2 + 7 \cdot 10^2 + 9 \cdot 10 + 2 \\ &= 6 \cdot 10^4 + 16 \cdot 10^3 + 17 \cdot 10^2 + 9 \cdot 10 + 2 \\ &= 60\,000 + 16\,000 + 1\,700 + 90 + 2 \\ &= 77\,992 \end{aligned}$$

Q3). What is the time complexity of that recursive D&C algorithm?

There are 4 multiplications of polynomials of degree $\frac{n}{2}$ and we need $O(n)$ work to combine the result.
$T(n) = 4 \cdot T(\frac{n}{2}) + O(n)$.
We can use Master theorem (case 1) and get $T(n) \in \Theta(n^2)$.
This is no better than the naive complete search polynomial multiplication.

Q4). Introducing: the Karatsuba's algorithm.
We still compute two smaller polynomials: $A_1(x) \times B_1(x)$, $A_2(x) \times B_2(x)$.
But instead of computing: $A_1(x) \times B_2(x), A_2(x) \times B_1(x)$ that requires two polynomial multiplications, we compute: $[A_1(x) + A_2(x)] \times [B_1(x) + B_2(x)]$ that requires two additions and just one multiplication.
Note: $A_1(x) \times B_2(x) + A_2(x) \times B_1(x) = [A_1(x) + A_2(x)] \times [B_1(x) + B_2(x)] - A_1(x) \times B_1(x) - A_2(x) \times B_2(x)$.
We now have the elements needed to compute $C(x)$ in faster time.

What is the time complexity of Karatsuba's algorithm?

There are only 3 multiplications of polynomials of degree $\frac{n}{2}$ and we still need $O(n)$ additional work.
$T(n) = 3 \cdot T(\frac{n}{2}) + O(n)$.
We can use Master theorem (still case 1) and get $T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.58})$.
It is built-in inside Python for multiplying two Big Integers.
PS: We can actually do polynomial multiplication in $O(n \log n)$, using a more advanced algorithm.

Q5). Decision Tree

You are given 243 balls, all but one of which have the same weight; the remaining one is heavier. Your job is to find which of the balls is heavier. Your friend has a balance scale, but will charge you for each weighing. You want to minimize the (worst-case) number of weighings needed. What is the minimum

number of weighings needed to find the ball?

You can assume that we only use comparison model (comparison returns $<$, $=$, or $>$). You can decide how to compare the ball(s). What is the lower bound of any algorithm to solve this problem?

Let $x = 243$.

Divide the balls into 3 equals groups: A, B, C, and weigh A against B, there are two possible outcomes:

1. If $A = B$, then the heavier ball is definitely in group C
2. Otherwise, if $(A > B)$, then the heavier ball is in group A; otherwise it is in group B.

Now notice that each weighing can decrease the size of the problem by $\frac{1}{3}$.

$$243 \xrightarrow{1\text{st}} 81 \xrightarrow{2\text{nd}} 27 \xrightarrow{3\text{rd}} 9 \xrightarrow{4\text{th}} 3 \xrightarrow{5\text{th}} 1.$$

The last ball left must be the answer, i.e., the heavier ball. Thus, we can find the heavier ball after 5 weighings.

To see that this is optimal, note that in each weighing, we can divide the balls into at most 3 groups. Thus, any algorithm using only the scale can be described as a decision tree, where each node has at most (because some weighings may not divide the balls into three groups) 3 children. Now, a full **ternary** tree of height 4 has only $3^4 = 81$ leaves, and we need to cater for the possibility that any of the 243 balls (leaves) being the heavier ball. Thus, 4 weighing cannot be enough.

Q6) You are given an array $A[1..n]$ that is sorted in **non-increasing order**. Your task is to find the largest index i such that $A[i] \geq i$. Design an efficient algorithm to solve this problem.
To guide your approach, consider the following properties of the sorted array:

- If $A[j] \geq j$, then it must hold that $A[j-1] \geq j-1$, unless $j = 0$.
- If $A[j] < j$, then it must follow that $A[j+1] < j+1$, unless $j = n$.

For ease of notation, assume that the array is extended such that $A[0] > 0 \quad \text{and} \quad A[n+1] < n+1$. Thus, there is a unique i such that $A[i] \geq i$ but $A[i+1] < i+1$.

Method 1: Do a linear search to find the largest i such that $A[i] \geq i$. Take $O(n)$ time.
Method 2: Do a binary search, taking advantage of the note mentioned above. Will take $O(\log n)$ time.
Method 3: First search for the least k such that $A[2^k] < 2^k$, by trying $k = 0, 1, 2 \ldots$. If $k = 0$, then we are already done.
Otherwise, this would give that $A[2^k] < 2^k$, but $A[2^{k-1}] > 2^{k-1}$. Now we can use binary search between $i = 2^{k-1}$ and 2^k, to get the largest i such that $A[i] \geq i$. This takes $O(\log i)$ time, where i is the final answer.
Note: This problem has several uses, such as H-index, see: H-index

Q7) Bogosort is an extremely inefficient sorting algorithm. It repeatedly generates random permutations of the input array until it encounters one that is sorted by chance. What is the best-case, worst-case and average-case time complexity of Bogosort for an array of length n?

Algorithm 1: Bogosort($A[0..n-1]$)

```
1 while not IsSorted(A) do
2     RandomlyShuffle(A)
3 return A
4 Function IsSorted(A):
5     for i ← 1 to n − 1 do
6         if A[i] < A[i − 1] then
7             return false
8     return true
```

Note: The RandomlyShuffle function can be implemented in $O(n)$ time using the Fisher-Yates shuffle algorithm.

- **Best-case complexity**: The best case occurs when the input array is already sorted. In this case, the algorithm only performs a single call to IsSorted, which takes $O(n)$ time, and no shuffling is needed.
- **Worst-case complexity**: The worst case is unbounded. Since each shuffle is random and independent, there is no guarantee that the algorithm will ever produce the correct permutation. In theory, the algorithm could run indefinitely.
- **Average-case complexity**: We assume that all possible permutations of the array are equally likely after each shuffle and that each shuffle is independent of previous attempts.

  For an array of length n, there are $n!$ possible permutations. Since each shuffle generates a random permutation, the probability of obtaining the correct permutation in a single shuffle is $\frac{1}{n!}$. The expected number of shuffles required to achieve a sorted order is $n!$.

  Each iteration consists of: RandomlyShuffle: $O(n)$, and IsSorted(A): $O(n)$. Thus, the total expected runtime is $O(n \cdot n!)$.