

ADA lecture notes

Analysis and design of algorithms

These are the notes for analysis and design of algorithms course. The professor says it'll be an interesting course, let's see about that. I am using Obsidian and this is an amazing markdown editor! It has a lot of community plugins. Anyways, study now... xD

Here is a somewhat detailed overview.

1. ADA/Lecture 1 : Introduction to the course and grading.
2. ADA/Lecture 2 : Mastering Master Theorem
3. ADA/Lecture 3 : DAC

Lecture->1

It's like DSA version 2 (in terms of management). Here's the link for previous year: ADA2020, ADA2022. Solutions and questions in this course are made by the instructor and hence making it public is not a good idea. So, these notes with stick around the lectures and maybe sometimes touching things but WILL NOT quote.

Evaluation

- Quizzes : 15% (n-1)
- Homework Assignments (Theory) : 15% (group of two)
- Programming Assignments : 10% (Foobar, No lab hours, Individual)
- Midsem : 30%
- Endsem : 30% Both theory

Multiplying large integers

Input : Two n -digit numbers A and B Output: Product $A \times B$ Primitive Ops: Add/Multiply two single digit integers (recall digital circuits adder)

- Classical pen-paper approach:
 - At max $2n$ operations per partial product, since n , we have $2n^2$
 - Summation of them, $2n^2$
 - Net $4n^2$
- Doing it differently: (Main idea: $\frac{n}{2}$ digits for each a, b, c, d)
 - $\overbrace{5678}^a \times \overbrace{1234}^c$
 1. Compute $a.c = 672$
 2. Compute $b.d = 2652$
 3. Compute $(a+b)(c+d) = 6164$
 4. Compute $\boxed{3.-2.-1.} = 2840$
 5. Put it all together $6720000 + 2652 + 284000$ (Notice the padding)
 6. Do it all recursively
 - Here's the recursive implementation, where $A = 10^{\frac{n}{2}} \cdot a + b, B = 10^{\frac{n}{2}} \cdot c + d$ and $A \times B = 10^n ac + 10^{\frac{n}{2}}(ad + bc) + bd$
 1. Recursively compute $a.c$
 2. Recursively compute $b.d$
 3. Recursively compute $(a+b) \cdot (c+d)$ (Karatsuba method, otherwise 4 recursive calls)
 4. Compute $\boxed{3.-2.-1.}$ for each call
 5. Pad and add!

Lecture->2

Analysis: The recurrence method

$T(n)$ = Runtime of Algorithm 1 for multiplying two n -digit numbers

Base Case : $n = 1, T(n) = c$: Multiplying two single digit numbers

Recurrence (for $n > 1$) : Express $T(n)$ as the runtime of recursive calls + additional work which maybe done in that call.

Recursively compute each ac, bd, bc, ad , work in this step

$$\begin{array}{rcl}
 T(n) & = & \overbrace{4T\left(\frac{n}{2}\right)}^{\text{computing ac,bd,bc,bd}} + \overbrace{c_1 \cdot n}^{\text{Adding 4 n/2 (+padded) digit no.s}} \\
 T(1) & & = \underbrace{c}_{\text{Base case}}
 \end{array}$$

Karatsuba's Algorithm (more like optimization) ($a + b$ may have $\frac{n}{2} + 1$ digits)

$$\begin{array}{rcl}
 T(n) & = & \overbrace{3T\left(\frac{n}{2}\right)}^{n/2+1 \text{ but ignore}} + \overbrace{(c_2 + c_3) \cdot n}^{\text{Adding a+b, multiplying each other}} \\
 T(1) & & = \underbrace{c}_{\text{Base case}}
 \end{array}$$

Master Method / Master Theorem

A 'Black-box' method to solve many common recurrences in Algorithm design (especially DAC)

Assumption: All the recursive calls are made on subproblems of equal size. (If not, use the proof we will do now)

Assumption (for proof): Both constants c are equal.

$$\begin{array}{lcl} T(n) & = & aT\left(\frac{n}{b}\right) + c \cdot n^d \\ T(1) & \leq & c \end{array}$$

$$a \geq 1, b \geq 1, c, d \geq 0$$

a = Number of recursive calls

b = Shrinkage factor of subproblem size

d = Affects the runtime of the additional work (outside recursion)

Master Theorem (Simpler Version): Prof. says no need to *remember*

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

Example: Mergesort, $T(n) = 2T(n/2) + c \cdot n$ so $a = 2, b = 2, d = 1$, so case 1. $T(n) = \mathcal{O}(n \log n)$

Example: Binary search, $T(n) = T(n/2) + c \cdot n^0$, so $a = 1, b = 2, d = 0$, so case 1. $T(n) = \mathcal{O}(\log n)$

Example: Multiplication algorithm1, $T(n) = 4T(n/2) + c_1 n$, so $a = 4, b = 2, d = 1$, so case 3. $\mathcal{O}(n^{\log_4 2})$

Example: Multiplication algorithm (Karatsuba), $T(n) = 3T(n/2) + c_1 n$, so $a = 3, b = 2, d = 1$, so case 3. $\mathcal{O}(n^{\log_3 2})$

- The calculator uses Strassen Schonenhage: $\mathcal{O}(n \log n \log \log n)$
- New proposed solution: $\mathcal{O}(n \log n)$

Proof of master theorem (Simpler version)

Assumptions:

1. n is a power of b (the shrinkage factor)
2. Base case: $T(1) = c$ (same as n^d)

Main technique:

- Recursion Trees (eww)
 - Levels: $\boxed{\log_b n + 1}$

- Subproblems at level j : a^j
- Subproblem size at level j : $\frac{n}{b^j}$
- *Total* work done outside recursive calls at level j : $a^j \cdot \left(\frac{n}{b^j}\right)^d \cdot c = n^d \left(\frac{a}{b^d}\right)^j \cdot c$
Should be intuitive from the above equation, Good = a , bad = b^d , in the end we just sum it all.
- So, work is sum of total work across all levels

$$\sum_{j=0}^{\log_b n + 1} n^d \left(\frac{a}{b^d}\right)^j \cdot c$$

Lecture->3



Divide and Conquer Algorithms

1. Divide (break into several parts)
2. Conquer (Solve the smallest solvable)
3. Combine (subproblems)

Counting Inversions in an Array

Input: 1, 3, 5, 2, 4, 6

Output: 3

Inversion pairs: (3, 2), (5, 2), (5, 4) {kind of like bubble sort}

Golden Benchmark to get inversions: Sorted array (ascending)

Trivial algorithm = $\mathcal{O}(n^2)$

Today: $\mathcal{O}(n \log n)$

Q. Can we output all inversions in same time above? No, total $O(n^2)$ possible invs.

- Key Ideas

- Suppose A is divided in to X and Y (possibly in the middle)
- An inversion pair (i, j) is :
 1. Left inversion : Both (i, j) in X
 2. Right inversion : Both (i, j) in Y
 3. Split inversion : i in X and j in Y
- Using recursion get (1), (2) and after you get the results, count split inversions.
- Here's the pseudo code

```
CountInv(array& A, length n):
    if n==1 return 0
    X = A[1,2,...n/2], Y=A[n/2+1,...,n]
    x = CountInv(X,n/2)
    y = CountInv(Y,n/2)
    x = CountSplitInv(X,n/2)
    return x+y+x
```

Copy

- Now, since split inversions wont be affected if we sort each X and Y (along the recursive calls) it'll get easier to count inversions. We will use this to count split inversions while merging. `count += (n/2 - i + 1)` where i is the iterator of X and we are using the combine/merge function. This takes advantage of sorting.