Algorithm Analysis Framework

Calculating the running time of an algorithm

Example: Finding max element of an array A of size n.

Algorithm: arrayMax (A, n)

Inputs: Array A[ ], size of array n

Output: Max element of A[ ]

1. currentMax ← A[1]; ⟶ 2 ops (indexing + assignment)
2. **for** i ← 2 to n **do** ⟶ 1 op (assignment) + 'n' ops (comparisons) + (n-1)*2 ops (increment + assignment)
3. **if** currentMax < A[i] ⟶ (n-1)*2 ops (indexing + comparison)
4. **then** currentMax ← A[i] ⟶ <=(n-1)*2 ops (indexing + assignment)
5. **return** currentMax ⟶ 1 op (return)

Total ops $T(n) = 2 + 1 + n + 2(n-1) + 2(n-1) + 2(n-1) + 1$
$= 7n - 2$

This is worst case when step 4 is executed every time. i.e. the list in sorted (ascending) order.

The best case happens when A[1] is the largest.

$$T(n)_{[\text{best case}]} = 7n-2 - 2(n-1) = 5n$$

The algorithm will run faster for some inputs than it does for others. Is there an average case? What is "average"? A tricky question. Need probability. Therefore, except for experimental studies or when we are sure that algorithm is itself randomized, we will typically characterise running time in terms of the worst case.

Several points are in order

1. In the above algorithm, we used index from 1, not from 0.
2. We assumed all kinds of operations take equal time. i.e. indexing, assigment, increment, return all takes the same time. In reality, it is not so.
3. The algorithm is designed considering single CPU architecture. (with infinite memory)
4. The algorithm is written using pseudo code.

These conventions are followed throughout the course.

To summarize, we use

- worst case time to characterize an algorithm
- average case for few scenarios (randomized, experiments)
- Amortization or amortized analysis for special cases (when there is a high frequency of low-cost operation and low frequency of high-cost operations)

  Ex: An implementation of dynamic array where we double the size of the array each time it fills up.

  More about this is dealt later in the course.

~~Hw~~ Exercises

Find the best case and worst case running times for the following problems. Write an algorithm & compute.

1) Search for an element x in an array of size n
2) Check if all the elements in an array are unique. i.e. there are no elements repeated
3) Matrix addition
4) Matric multiplication

Analysing Recursive Algorithms

Let us consider the recursive version of the arrayMax algorithm.

Algorithm: recursiveMax (A[], n)

Input: Array A with $n \geq 1$ integers

Output: Max element in A[]

(1) if (n = 1)

(2) then return A[1]

(3) else return max (recursiveMax (A[], n-1), A[n])

Recursive algorithms are often quite elegant. How do we compute the running time of such algorithms?

It takes a bit of additional work. We use recurrence equation which defines mathematical statements that the running time must satisfy.

We can characterize the running time T(n) of recursiveMax algorithm as:

$$T(n) = \begin{cases} 3, & \text{if } n=1 \quad [\text{comparison + indexing + return}] \\ T(n-1)+7 & \text{otherwise} \end{cases}$$

$\Downarrow$

(3) return max (recursiveMax (A[], n-1), A[n])

return	max	recursiveMax(A[], n-1)	A[n]
1 op [return]	3 ops [comparison + max call + max return]	T(n-1) + 1 op [recursive call]	1 op (indexing)

Add 1 op from step (1) [if (n = 1)

∴ Total $T(n) = T(n-1) + 7$

This eqn is called as recurrence eqn.

How do we solve this recurrence eqn to a closed form?

1 Substitution method

$$
\begin{aligned}
T(n) &= T(n-1) + 7 \\
&= T(n-2) + 2(7) \\
&= T(n-3) + 3(7) \\
&\;\vdots \\
&= T(1) + (n-1)(7) \\
&= 3 + 7n - 7 \quad [\text{3 ops for the base case}] \\
T(n) &= 7n - 4
\end{aligned}
$$

In general, determining closed form solutions to recurrence equations can be much more challenging.

2 Guess and prove by Induction

Lets say we were able to guess the closed form (not always easy)

$$T(n) = 7n - 4$$

We try to prove by induction

Base case: $n = 1 \Rightarrow T(n) = 7(1) - 4 = 3$

Matches with the recurrence eqn

Inductive hypothesis: The eqn is true for some m.

Now, we have to prove that the eqn holds for $(m+1)$ too.

i.e Assume $T(m) = 7m - 4$ and derive eqn for $T(m+1)$ in the same way

$$\cancel{T(m+1) = 7(m+1) - 4 = 7m + 7 - 40}$$

$$\cancel{= T(m) + 7}$$

$$T(m+1) = 7(m+1) - 4 = 7m + 3 \quad — 1$$

$$T(m+1) = T(m) + 7 = 7m - 4 + 7 = 7m + 3 \quad — 2$$

1 + 2 are the same. Hence proved.

Lets do some more examples

Ex2: Finding factorial of n (using recursion)

Algorithm: To find factorial (n)

Input: A number $n \geq 1$

Output: $n!$

① if $n = 1$ $\longrightarrow$ 1 op

② then return 1 $\longrightarrow$ 1 op

③ else $\underbrace{\text{return}}_{\text{1 op (return)}}$ $\underbrace{\text{factorial}(n-1) * n}_{T(n-1) \,+\, \text{1 op (multiplication)}}$

Base case: $T(1) = 2$ [steps ① + ②]

Other cases: $T(n) = T(n-1) + 3$ [steps ① + ③]

Substitution Method

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= [T(n-2) + 3] + 3 = T(n-2) + 2(3) \\
&= T(n-3) + 3(3) \\
&= \vdots \\
&= T(1) + (n-1)(3) \\
&= 2 + 3n - 3
\end{aligned}
$$

$T(n) = 3n - 1 \longrightarrow$ closed form eqn

Lets prove by induction

$T(1) = 3(1) - 1 = 2$ which matches the base case

Assume the recurrence eqn holds for some m.

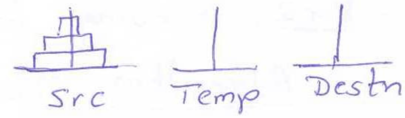
We need to prove it holds for $(m+1)$ too.

$$
\begin{aligned}
T(m+1) = T(m) + 3 = 3m - 1 + 3 &= 3m + 3 - 1 \\
&= \underbrace{3(m+1) - 1}_{\text{same for as we expected.}}
\end{aligned}
$$

Hence, proved.

Ex 3: Tower of Hanoi (recursive algorithm)

- Move discs from src to Destn using Temp as intermediate 1-by-1
- Always ensure smaller disk ~~is~~ above larger disc.



Algorithm: TowerOfHanoi (n, Src, Temp, Destn)

Input: Src, Temp, Destn with Src containing 'n' disks.

Output: Destn containing the discs in same order.

1. if $(n = 1)$
2. then Move disk from src to Destn
   else
3. TowerOfHanoi ($n-1$, Src, Destn, Temp)
4. Move disc n from Src to Destn
5. TowerOfHanoi ($n-1$, Temp, Src, Destn)

Recurrence equation

$$T(n) = \begin{cases} 2 & \text{if } n = 1 \quad \text{(base case)} \\ 2T(n-1) + 2 & \text{if } n > 1 \quad \text{(other cases)} \end{cases}$$

substitution method

$$\begin{aligned} T(n) &= 2T(n-1) + 2 \\ &= 2[2T(n-2) + 2] + 2 = 2^2 T(n-2) + 2^2 + 2 \\ &= 2^3 T(n-3) + 2^3 + 2^2 + 2 \\ &= 2^{n-1} T(1) + 2^{n-1} + 2^{n-2} + \ldots + 2 \\ &= 2^{n-1} \cdot 2 + 2^{n-1} + 2^{n-2} + \ldots + 2 \\ &= 2^n + 2^{n-1} + \ldots + 2 \end{aligned}$$

$$\text{i.e. } T(n) = 2^{n+1} - 2 = 2(2^n - 1)$$

n	1	2	3	4	5
$T(n)$	2	6	14	30	62

$n = 3 \Rightarrow T(n) = 2^3 + 2^2 + 2 = 14 = 2^4 - 2$

$n = 4 \Rightarrow T(n) = 2^4 + 2^3 + 2^2 + 2 = 30 = 2^5 - 2$

$n = 5 \Rightarrow T(n) = 2^5 + 2^4 + 2^3 + 2^2 + 2 = 62 = 2^6 - 2$

Hence, verified

Exc4: Digits in a binary no. given a decimal number

0, 1 → single binary digit (0 to 2^1-1)

2, 3 → Double binary digits (2^1 to 2^2-1)

4, 5, 6, 7 → 3 binary digits (2^2 to 2^3-1)

⋮

Algorithm: Binary (n)

Inputs: A positive integer n

Output: No. of digits in binary representation of n.

if n = 1 → 1 op

then return 1 → 1 op

else return 1 + Binary(n/2) → 2 ops + T(n/2)

Recurrence eqn

$$T(n) = \begin{cases} 2 & \text{if } n \le 1 \\ 2 + T(n/2) & \text{otherwise} \end{cases}$$

Substitution method

$$\begin{aligned} T(n) &= T(n/2) + 2 &&= T(n/2) + 2\cdot 1 \\ &= T(n/4) + 2 + 2 &&= T(n/2^2) + 2\cdot 2 \\ &= T(n/8) + 2 + 2 + 2 &&= T(n/2^3) + 2\cdot 3 \\ &= T(n/16) + 2+2+2+2 &&= T(n/2^4) + 2\cdot 4 \\ &\vdots \\ &= T\left(n/2^{\log n}\right) + 2\cdot \log n \end{aligned}$$

[Note: $2^{\log n} = n$]

$$\begin{aligned} &= T(1) + 2\log n \\ &= 2 + 2\log n \\ &= 2(1+\log n) \end{aligned}$$

The table below summarizes our analysis:

Problem	Time Taken (worst case)
1) Helloworld	1 (constant)
2) Finding if x exists in A[ ] (unsorted)	n (linear)
3) Finding if x exists in A[ ] (sorted)	$\log n$ (logarithmic)
4) Finding max element in A[ ] (unsorted)	$n + \log n - 2$ (2^{nd} algorithm) (or) $2n - 3$ (1^{st} algorithm)
5) Finding GCD	(Varying) $\sqrt{n}$ (square root)
6) Finding max element in A[ ] (unsorted)	$7n - 2$ (iterative) or $7n - 4$ (recursive)
7) Finding factorial	$3n - 1$ (linear)
8) Tower of Hanoi	$2^{n+1} - 2$ (exponential)
9) Finding no. of binary digits	$2(1 + \log n)$ (logarithmic)
10) Traveling salesman problem n cities/vertices	$n!$ (factorial)
11) Matrix addition ($n \times m$)	nm (quadratic)
12) Matrix multiplication ($n \times m$)	$\left(\frac{n}{2}\right)^{3/2}$ (cubic)

In general, $T(n)$ takes the form

$$T(n) = a\, f(n) + b$$

where $f(n)$ can be $n, n^2, \log n, n\log n, \sqrt{n}, n^3, 2^n, n!, \ldots$

As n increases, a and b becomes insignificant. Hence, we consider only $f(n)$ for our analysis and leave out the constants.