CS3230 (Analysis of Algorithms)

Jia Cheng

September 2021

1 Definitions and Formula

Sum of geometric progression

$$\sum_{0 \le i \le n} r^i = \frac{r^0 - r^{n+1}}{1 - r} = \frac{1 - r^{n+1}}{1 - r}$$

Importantly, remember that the top term is $1 - r^{n+1}$, not $1 - r^n$!!!!!!

In general, for a geometric progression $a_1, a_2, \ldots, a_n$ with common ratio r, its sum is

$$\frac{a_1 - a_n \cdot r}{1 - r}$$

Sum of arithmetic progression For an arithmetic progression $a_1, a_2, \ldots, a_n$ with common difference d, its sum is

$$\frac{n \cdot (a_1 + a_n)}{2}$$

Decision tree Three terms **level**, **depth**, **height**

```
// I've used 1 for roots level
// though some people consider roots level as 0, so you can use either 0 or 1
// I would prefer to use 1
// but its your choice

                                o(depth=0, height=3, lev=1)
                               /   \
     (depth=1, height=2, lev=2)o     o(depth=1, height=1, lev=2)
                              /     / \
    (depth=2, height=1, lev=3)o     o   o(depth=2, height=0, lev=3)
                             /
  (depth=3, height=0, lev=4)o
```

Image from `https://stackoverflow.com/questions/63369552/depth-vs-level-of-a-tree`

- For a complete binary tree, number of children at depth k is 2^k. In particular, the number of children at depth 0 is $2^0 = 1$ (the root node).
  For example, in a comparison based sorting algorithm, the depth equals to the number of comparisons made.

- Height = Maximum depth of tree
- Level = Depth + 1
- Usually, depth is more useful than level since depth is 0-indexed

Asymptotic notation From wikipedia

Notation	Name[24]	Description	Formal definition	Limit definition[25][26][27][24][19]
$f(n) = O(g(n))$	Big O; Big Oh; Big Omicron	$\|f\|$ is bounded above by g (up to constant factor) asymptotically	$\exists k > 0\, \exists n_0\, \forall n > n_0 \colon \|f(n)\| \leq k \cdot g(n)$	$\limsup_{n\to\infty} \frac{\|f(n)\|}{g(n)} < \infty$
$f(n) = \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$\exists k_1 > 0\, \exists k_2 > 0\, \exists n_0\, \forall n > n_0 \colon$ $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Knuth version)
$f(n) = \Omega(g(n))$	Big Omega in complexity theory (Knuth)	f is bounded below by g asymptotically	$\exists k > 0\, \exists n_0\, \forall n > n_0 \colon f(n) \geq k \cdot g(n)$	$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) = o(g(n))$	Small O; Small Oh	f is dominated by g asymptotically	$\forall k > 0\, \exists n_0\, \forall n > n_0 \colon \|f(n)\| < k \cdot g(n)$	$\lim_{n\to\infty} \frac{\|f(n)\|}{g(n)} = 0$
$f(n) \sim g(n)$	On the order of	f is equal to g asymptotically	$\forall \varepsilon > 0\, \exists n_0\, \forall n > n_0 \colon \left\|\frac{f(n)}{g(n)} - 1\right\| < \varepsilon$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$
$f(n) = \omega(g(n))$	Small Omega	f dominates g asymptotically	$\forall k > 0\, \exists n_0\, \forall n > n_0 \colon \|f(n)\| > k \cdot \|g(n)\|$	$\lim_{n\to\infty} \frac{\|f(n)\|}{g(n)} = \infty$
$f(n) = \Omega(g(n))$	Big Omega in number theory (Hardy–Littlewood)	$\|f\|$ is not dominated by g asymptotically	$\exists k > 0\, \forall n_0\, \exists n > n_0 \colon \|f(n)\| \geq k \cdot g(n)$	$\limsup_{n\to\infty} \frac{\|f(n)\|}{g(n)} > 0$

Upper and lower bounds In algorithmic analysis, when we speak of upper and lower bounds, we are usually referring to the worst case. Of course, we tend to use bounds because we cannot quantify precisely what is the running time of an algorithm over all inputs.

For instance, if we know for a certain n, the runtime of algorithm M over all inputs of size n is $\{n, n+1, \ldots, 2n-1, 2n\}$, then $[2n, \infty)$ would all be valid upper bounds, and $[0, 2n]$ would be valid lower bounds. However, since we already know the worst case runtime: $2n$, there is no point talking about bounds.

Upper bounds require universal quantification. That is, to say that $U(n)$ is an upper bound on the runtime, we need $\forall$ inputs of size n, runtime is $\leq U(n)$.

In comparison, existential quantification is sufficient to justify a lower bound. To say that $\exists$ input of size n, runtime $\geq L(n)$, we simply need to provide an instance of an input that indeed results in $L(n)$ or more runtime.

The above can be described in terms of order theory. Let $S(n)$ be a set (representing the runtimes of size n inputs).
The worst case runtime is analogous to $\sup S(n)$.

For a n dependent value $U(n)$, $\sup S(n) \leq U(n) \iff \forall s \in S(n), s \leq U(n)$.

For a n dependent value $L(n)$, $\sup S(n) \geq L(n) \iff \exists s \in S(n), s \geq L(n)$. This should be a very familiar statement in mathematical analysis. In particular, if $\exists s \in S(n), s = L(n)$, then indeed $L(n) \leq \sup S(n)$.

In conclusion, this asymmetry in algorithmic analysis is due to the fact that we only care about supremums and not infimums.

2 Bounding Techniques

Integral Bound For a monotonically increasing function f,

$$\int_{i-1}^{j} f(x)\,dx \leq \sum_{x=i}^{j} f(x) \leq \int_{i}^{j+1} f(x)\,dx$$

For a monotonically decreasing function f.

$$\int_{i}^{j+1} f \leq \sum_{x=i}^{j} f(x) \leq \int_{i-1}^{j} f$$

Applications

- Harmonic series
  $H_n = \sum_{1\leq i\leq n} \frac{1}{i} = 1 + \sum_{2\leq i\leq n} \leq 1 + \int_1^n \frac{1}{i}\,di = 1 + \ln(n) = O(\log n)$
  $H_n \geq \int_1^{n+1} \frac{1}{i}\,di = \ln(n+1) = \Omega(\log n)$
  such that $H_n = \Theta(\log n)$

Telescoping Suppose we have the recursion $T(n) = aT(n/a) + nf(n)$. Then

$$\frac{T(n)}{n} = \frac{T(n/a)}{n/a} + f(n)$$

and we can telescope on $S(n) = \frac{T(n)}{n}$

Stirling's Approximation

$$f(n)! \sim \sqrt{2\pi f(n)}\left(\frac{f(n)}{e}\right)^{f(n)}$$
$$\ln(f(n)!) \sim f(n)\ln(f(n)) - f(n) = \Theta(f(n)\ln(f(n)))$$

Logarithms

$$\lim_{n\to\infty} \frac{\log(\log(n)-k)}{\log(\log(n))} = \lim_{x\to\infty} \frac{\ln(x-k)}{\ln(x)}$$
$$= \lim_{n\to\infty} \frac{\frac{1}{x-k}}{\frac{1}{x}} = 1$$

Hence $\log(\log(n)-k) \sim \log(\log(n))$

In general, define $f^n = f \circ f^{n-1}$, and

$$\lim_{x\to\infty} \frac{\ln^n(x-k)}{\ln^n(x)} = \frac{\frac{1}{\ln^{n-1}(x-k)} \cdot \frac{1}{\ln^{n-2}(x-k)} \cdots \frac{1}{\ln(x-k)} \cdot \frac{1}{x-k}}{\frac{1}{\ln^{n-1}(x)} \cdot \frac{1}{\ln^{n-2}(x)} \cdots \frac{1}{\ln(x)} \cdot \frac{1}{x}}$$

We can use strong induction to show that this limit is 1.

Arithmetic sequence-like summations Consider a summation of the form

$$\sum_{0 \leq k \leq \frac{n}{m}} f(n - mk)$$

where m is a positive constant, and f is a monotonically increasing function. If we expand out the summation, we realize that approximately $\frac{n}{2m}$ of the terms are greater than $\frac{n}{2}$.

To see this,

$$n - mk \geq \frac{n}{2} \iff \frac{n}{2} \geq mk \iff k \leq \frac{n}{2m}$$

Hence, we can consider the following **lower** bound

$$\sum_{0 \leq k \leq \frac{n}{m}} f(n - mk) \geq \sum_{0 \leq k \leq \frac{n}{2m}} f(\frac{n}{2}) \approx \frac{n}{2m} \cdot f(\frac{n}{2})$$

Usually such a bound is much tighter than simply taking the last term, $f(n)$, since we are taking into account a larger number of terms in the sum.

The no-brainer **upper** bound for such a sequence would be

$$\sum_{0 \leq k \leq \frac{n}{m}} f(n - mk) \leq \sum_{0 \leq k \leq \frac{n}{m}} f(n) \approx \frac{n}{m} f(n)$$

Decision Tree The degree of branching that a decision tree based algorithm allows depends on the number of outcomes per step.
To be precise, suppose each step of an algorithm is able to partition the set of valid inputs into k disjoint sets. Then we can think of this as branching factor k.

For e.g. if a single step in an algorithm allows 3 different types of outcomes, then the decision tree is ternary.

Suppose we can have n possible results in total.
A decision tree with branching factor k and height (maximum depth) h has the following properties. Let L be the number of leaves of the tree.

- The leaves represent the outcomes. $L \leq 2^h$. Equality holds when the tree is complete.
  Example: A comparison based sorting algorithm has a branching factor of 2. Hence if h comparisons are made, the tree has at most 2^h leaves. A necessary condition for the sorting algorithm to be correct (when there are n elements) is that $L \geq n!$. Hence $2^h \geq n!$ and we have the familiar bound $h \geq \lg n!$
- The height of the tree represents the number of operations/"steps" that are taken. E.g. When 0 operations are taken, the tree is of height 0 and it only consists of 1 leaf which is also the root node. Hence there are $2^0 = 1$ leaves.

When are decision trees ineffective in analysis?

- Decision trees are good when **distinct inputs give distinct answers**. In other words, a correct algorithm is necessarily injective. Due to this property, if a decision tree is unable to produce as many leaves as the possible range of inputs, i.e. at least 1 leaf encompasses > 1 input, then **the algorithm is unable to distinguish 2 distinct inputs with distinct solutions** and hence must be incorrect.

- An example of incorrect usage of decision tree would be a simple query model problem. Taken from lecture 1,
  In the string query model, the input is a string of n bits. In one time unit, an algorithm can query one bit of the string. All other operations on the string can be performed at 0 cost. Let Zero be the problem of deciding whether an input string x is the all-zero string or not. The optimal algorithm for Zero has query complexity n.
  There are 2^n valid inputs, since each entry $A[i]$ of an array A can either be 1 or 0. Additionally, the decision tree is a binary one, since when the algo queries index $i \in \{1, \dots, n\}$, we either reply $A[i] = 1$ or $A[i] = 0$.
  However, even if the height of the decision tree is strictly less than n, such that there are strictly fewer than 2^n leaves, **we cannot immediately claim that the algo is incorrect**. Because it is certainly possible that the input $A[1] = A[2] = \cdots = A[n] = 0$ is in its own leaf, alone. The other leaves that encompass multiple inputs then all give the same answer (i.e. the string is not the All-Zero string). Then the adversarial argument would not work here.

Jensen's inequality for expected values

Let X be a random variable and let f be a function.
If f is concave (e.g. $x \mapsto \sqrt{x}$), then $E[f(X)] \leq f(E[X])$ (e.g. $E[\sqrt{X}] \leq \sqrt{E[X]}$) .
If f is convex (e.g. $x \mapsto x^2$), then $E[f(X)] \geq f(E[X])$.
It is due to the convexity of the square function that $Var(X) = E[X^2] - E[X]^2 \geq 0$

3 Distributions

Poisson approximation of Binomial Distribution Given binomially distributed $X \sim B(n, p)$, $X \approx Y \sim Poi(\lambda = np)$, where $Y \in \{0, 1, 2, \dots\}$

$$\text{p.m.f. of } Y\text{: } P(Y = k) = e^{-\lambda}\frac{\lambda^k}{k!}$$
$$P(Y = 0) = \exp(-\lambda)$$
$$P(X > 0) \approx P(Y > 0) = 1 - \exp(-\lambda)$$

4 Graphs

Suppose a graph has vertices V and edges E. Some common concepts associated with graphs (which we can use to argue or prove things are:

- V
- Pairs of points (V choose 2), adjacency matrix
- E
- Degree of vertex v, $deg(v)$
- Endpoints of edges
- Coloring nodes (this is a more general concept, and colors can often mean different things in different contexts)

Handshake Lemma

$$\sum_{1 \leq i \leq |V|} deg(v_i) = 2 \cdot |E|$$

5 Algorithms

Themes

Similar subroutines to well-known algorithms

- Quickselect
- Partition
- Merge
- Radix sort (e.g. when items to sort, usually integers, are bounded above)
- Linear search
- Binary search (e.g. Searching the maximal element in a sorted, then cyclically rotated array)

Paradigms

- Divide and conquer
  - Returning additional metadata (to improve asymptotic efficiency) e.g. O(n) version of maximal contiguous subarray sum/product
- Streaming (e.g. find majority element)

5.1 Dynamic Programming

First, we have a rigorous formulation of dynamic programming. A usual DP recursive formula takes the form of

$$dp(n) = C + \max\{dp(i) : i \in S\}$$

where $S \subseteq \{1, 2, \dots n-1\}$ is a set of subproblems to be considered and C is some constant. max can be replaced by any aggregate function, but we only consider maximization problems here for simplicity.

The usual proof of this equality involves the following conditions.

1. $\exists i \in S, dp(n) = C + dp(i)$
2. $\forall i \in S, dp(n) \geq C + dp(i)$

When we argue that "Given an optimal solution S_n to the problem with input n, S_n contains within it an optimal solution S_i, for some $i \in S$.", we are demonstrating condition (1).
Condition (2) can be show using cut and paste. A way to justify (2) is to show that every solution to a subproblem S_i can be built up to a solution S'_n with value $dp(i) + C$, and by optimality of S_n, S'_n must be inferior, such that $dp(i) + C \leq dp(n)$. Usually, cut and paste (in CS3230) is limited to showing optimality of a specific subproblem S_i and not the full extent of condition (2).

(1) implies that $dp(n) \leq C + \max\{dp(i) : i \in S\}$, since $dp(n) \leq C + dp(i)$.
(2) implies that $dp(n) \geq C + \max\{dp(i) : i \in S\}$. Hence we have the two way bound

$$C + \max\{dp(i) : i \in S\} \leq dp(n) \leq C + \max\{dp(i) : i \in S\}$$

which proves the recursive relation.

Greedy The greedy equivalent of the DP formula would be

$$g(n) = C + g(f(n))$$

where $f(n) \in \{1, 2, \ldots, n-1\}$ is a particular subproblem to be considered. f is the greedy choice function.

Greedy choice property: There exists an optimal solution making a particular choice, which can be understood as showing that we can define $f(n)$.
Optimal substructure (for this particular choice): $C + g(f(n)) \leq g(n) \leq C + g(f(n))$.

Subproblem Graph CLRS Pg 367, 380-381

Looking at the graph, the runtime of a memoized algorithm is the number of directed edges. (Notice that each edge is "traversed" exactly once). Hence, by summing the outdegrees of each vertex, we can obtain the overall runtime.

The edges represent calling the subprocedures, but not "stepping into" them.
The outdegree of a vertex is usually (but not always) the runtime of a single call to the procedure without further recursion.
If the procedure itself has something additional computation involved internally, then we will also need to factor this into the runtime. This can be represented by the vertex itself.
Combining the above 2 points, we need to add up the edges and the vertices.

Hence the runtime of a memoized procedure, say $func$ is $T(n) = O(\sum_{1 \leq i \leq n} S(i))$, where $S(i)$ is the runtime of a call to $func(i)$ without going into the recursive calls.

A loose upper bound Adapted from CLRS Pg 380 There are 2 factors that determine the running time of a DP solution

- (1) how many subproblems an optimal solution to the original problem uses,
- (2) how many choices we have in determining which subproblem(s) to use in an optimal solution

A (possibly loose) upper bound for the DP solution would then be the product of (1) and (2).

5.2 Network flow

Terminology

- Network Graph $G = (V, E)$ with capacity function $c = (u, v) \mapsto \mathbb{R}$
  - Simplication: Antiparallel property, $(u, v) \in E \implies (v, u) \notin E$
  - $(u, v) \notin E \implies c(u, v) = 0$
- Flow $f : V \times V \to \mathbb{R}$, $f = (u, v) \mapsto f(u, v)$ where $f(u, v)$ abides by 2 constraints
  - Capacity upper bound: $0 \leq f(u, v) \leq c(u, v)$
  - Conservation: $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$
- Value of a flow, $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Note that in a graph without antiparallel edges, the second term is always 0. However, we would also like to consider flow of residual graphs, which can have edges from vertices v to source s. Furthermore, an augmenting flow f' can have negative value since this does not go against the definition. (Of course, in practice, we do not make use of augmenting flows with non-positive value. In particular, this is always possible according to lemma 26.2 of CLRS.)

- Residual graph augmented by flow f, G_f
  - Residual capacity function c_f, given by $c_f(u,v) = \begin{cases} c(u,v) - f(u,v) \text{ if } (u,v) \in E \\ f(v,u) \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$
    In particular $c_f(u,v)$ is the residual capacity of an edge (u,v).
  - Edge set of residual graph $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$
  - Augmenting path $p = s \rightsquigarrow t$ in G_f
  - Residual capacity of path p, $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
  - Augmentation of flow f by f', $(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) \text{ if } (u,v) \in E \\ 0 \text{ otherwise} \end{cases}$

Note that lemma 26.2 states that for a path p in G_f, the flow f_p, defined by

$$f_p(u,v) = \begin{cases} c_f(p) \text{ if } (u,v) \in p \\ 0 \text{ otherwise} \end{cases}$$

is such that $|f \uparrow f_p| = |f| + |f_p| > |f|$ since $c_f(p) > 0$ (we only allow edges with positive weights in E_f).

6 Reductions

Let Σ, Γ be 2 alphabets. For 2 decision problems $A \subseteq \Sigma^*$, $B \subseteq \Gamma^*$, $A \leq_p B$ iff there exists a polynomial time computable function $f : \Sigma^* \to \Gamma^*$ such that $a \in A \iff b = f(a) \in B$.

This is a particular case of a more general concept of many-to-one reduction $\leq_m$.