1 Complexity and Master Theorem

Complexity. Symbols are $o, O, \Theta, \Omega, \omega$ for $<, \leq, \approx, \geq, >$, respectively. Recall that $f(n) = \text{symbol}(g(n))$ iff $\lim_{n\to\infty} f(n)/g(n)$ is 0, less than constant, constant, more than constant, ∞.

Formally, for big O, exists constant c, n_0 if $0 \leq \frac{f(n)}{g(n)} \leq c \forall n \geq n_0$.

Sometimes you need to explicitly write the constants.

Note: The definition of Θ that I made above is actually a bit inaccurate. $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $g(n) = O(f(n))$ (or, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, or something similar).

This implies that Θ means *for n sufficiently large*, there exists "a lower bound" and "an upper bound" such that

$$\text{lower bound} \leq \frac{f(n)}{g(n)} \leq \text{upper bound},$$

or, in other words, both

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0 \text{ and } \limsup_{n\to\infty} \frac{f(n)}{g(n)} < +\infty$$

must hold.

Moreover, $f(n) = O(g(n))$ iff the limsup property holds and $f(n) = \Omega(g(n))$ iff the liminf property holds.

Master Theorem. Given $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. crit $:= \log_b(a)$, ϵ, k, c constants > 0, ≥ 0, < 1.

- If ϵ, s.t. $f = O(n^{\log_b(a)-\epsilon})$, then $T(n) = \Theta(n^{\log_b(a)})$.
- If k, s.t. $f = \Theta\left(n^{\log_b(a)} \log^k(n)\right)$, $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$.
- If ϵ, s.t. $f = \Omega(n^{\log_b(a)+\epsilon})$, and f satisfy *regularity condition* $[af(n/b)/f(n) \leq c \ (< 1)$ for all n large$]$, $T(n) = \Theta(f(n))$.

Other than that, use **telescoping method**; i.e. expressing expression as $\frac{T(n)}{g(n)} = \frac{T(n/b)}{g(n/b)} + h(n)$, **guessing/substitution + induction** method, or **recursion tree**.

2 Correctness and DnC

Good Algorithm.

- Must be correct (include corner cases),
- efficient (economical in time, space, resources),
- well-documented with sufficient details,
- maintainable (not tricky, easy to modify and abstractions are easily understood, not computer-dependent, usable as modules by others).

Correctness of Iterative Algorithms. (Involves loop). *Key in reasoning is loop invariant*, which is supposed to be:

- true at the beginning of an iteration,
- and remains true after the iteration (i.e. at beginning of next iteration).

How to use said invariant to show correctness:

- Need to show three things ("Induction").
- **Initialization:** true before first iteration of loop,
- **Maintenance:** if true before an iter, remains true before next iter,
- **Termination:** when algo terminates, invariant has *useful property* for showing correctness.

Correctness of Recursive Algorithms. Use math induction on *size of problem*.

More specifically, use strong induction $\Rightarrow$ prove base cases and show algo works correctly assuming algo works correctly for **all** smaller instances/cases.

Divide-and-conquer design paradigm. Divide problem into subproblems, **conquer** subproblems by solving them recursively, **combine** subproblem solutions.

This often leads to efficient algorithms (note: except if it can be solved using "pure logic" e.g. Kadane's algo in Practice Set 1 #5.)

3 Sorting and Decision Trees

What is Decision Tree? Tree-like model, where

- every node is a comparison,
- every branch represents the outcome of node's comparison,
- every leaf represents "class label" (decision after all comparisons).
- Note: worst-case running time is *height of tree*.

Counting and Radix Sort. Counting sort algo:

Counting sort

Set C[i] be the number of elements equal i.

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1        ▷ C[i] = |{key = i}|
```

Set C[i] be the number of elements smaller than or equal i.

```
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]         ▷ C[i] = |{key ≤ i}|
```

Move elements equal i to B[C[i-1]+1..C[i]].

```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

Takes $\Theta(n+k)$ time to sort n numbers in range $[1, k]$.

Radix sort: suppose there are T "digits". Do sort by the t^{th} least significant digit ($t \leftarrow 1$ to T) using a stable sorting algo.

Suppose the n b—bit "words" (radix sort) is broken into r—bit pieces, each pass of counting sort takes $\Theta(n+2^r)$ time.

Since there are b/r passes, we have:

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right).$$

Choose r to minimize $T(n,b)$: as $r > \log(n)$, time grows exponentially in n; so pick r close to $\log(n)$ (slightly smaller) by differentiation.

Correctness of radix sort. Induction on digit position.

Note: radix sort is fast when number of passes is small. Downside is little locality of reference, and a well-tuned quicksort fares better on modern processors (which features steep memory hierarchies).

Conclusion. Each sorting algorithm can handle different issues, as follows.

- Memory issue: InPlace sort.
- Need to preserve original order of *equal elements*: Stable sort.
- Average case fast: QuickSort.
- Worst case fast (comparison model): worst case $O(n\log(n))$ time sort.
- Worst case fast (non-comparison model): radix sort.

4 Randomized Algoroithms, illustrated by QuickSort

QuickSort Divide and Conquer Algorithm.

- *Divide:* Partition array into two subarrays around a ***pivot*** x such that the elements in lower subarray $\leq x \leq$ elements in upper subarray.
- *Conquer:* Recursively sort two subarrays.
- *Combine:* Trivial.

Suppose the "pivot" produces the subarrays with size j and $(n-j-1)$, then

$$T(n) = T(j) + T(n-j-1) + \Theta(n).$$

Average-case analysis of QuickSort. Uses $A(n) :=$ average running time of QuickSort, input size n. (Input chosen uniformly at random from the set of $n!$ permutations.)

$A(n) \approx 1.39 n \log_2(n)$.

Serious problem with QuickSort: time taken depends on initial/input permutation (is real data random?). Can we make it distribution insensitive?

If we pick pivot uniformly at random, time taken depends on random choices of pivot elements.

What makes Randomized Algorithms so Popular? Answer, based on lecture slides:

What makes Randomized Algorithms so Popular? [A study by *Microsoft* in **2008**]

Title: Cycles, Cells and Platters: An Empirical Analysis of **Hardware Failures** on a Million Consumer PCs

Authors: **Edmund B. Nightingale, John R. Douceur, Vince Orgovan**

Available at : research.microsoft.com/pubs/144888/eurosys84-nightingale.pdf

Event	Probability
My desktop will **crash** during this lecture	$> 10^{-7}$
RandQsort takes time at least **double** the average	$< 10^{-15}$

$n \geq 10^6$

\+ Simplicity

Note: in the lecture, we sum over i and j from the perspective of e_i (i^{th} smallest element) and e_j, we have $\Pr[e_i \text{ and } e_j \text{ get compared}] = \frac{2}{j-i+1}$.

Types of Randomized Algorithms.

- **Las Vegas Algo.** Output always correct, runtime is a random variable.
- **Monte Carlo Algo.** Output may be incorrect with (very) small probability, runtime is deterministic.

5 Weirder Stuff (Hashing, Worst-case Lin-time Order Stats)

Universal Hashing. Given hash table T with m slots, stores n elements, the **load factor** $\alpha := n/m$ is the average number of elements in a chain.

The hash function h is a randomized procedure satisfying **universal hashing** assumption: for any pair of **distinct** keys u, v,

$$\Pr[h(u) = h(v)] \le \frac{1}{m}$$

where *probability is over the* **random** *choice of the hash function* h.

For a key u contained in T, expected length of the chain $h(u)$ is at most $1+\alpha$.

Reason: length of chain at $h(u)$ [key means the number being inserted] is

$$1 + \sum_{v \neq u, v \in T} C_{uv} (:= \text{int}(h(u) == h(v))),$$

so length of chain is "at least one" since u itself belong to u's chain.

Worst-case linear-time order statistics. "This is a ketok-magic algo that was not here in my semester" —Edbert.

Developing the recurrence

$T(n)$ **Select**(i, n)

$\Theta(n)$ 1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

$T(n/5)$ 2. Recursively **Select** the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

$\Theta(n)$ 3. **Partition** around the pivot x. Let $k = \text{rank}(x)$.

4. **If** $i = k$ **then** return x

$T(3n/4)$ 5. **Else if** $i < k$ **then** recursively **Select** the ith smallest element in the lower part

6. **Else** recursively **Select** the $(i-k)$th smallest element in the upper part

Solving the recurrence. Substitution: $T(n) \le cn$.

$$\begin{aligned} T(n) &\le T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n) \\ &\le \frac{19}{20}cn + \Theta(n) \\ &= cn - \left(\frac{1}{20}cn - \Theta(n)\right) \\ &\le cn. \end{aligned}$$

Choose c large enough to handle both $\Theta(n)$ and the initial conditions.

6 Tutorials and Examples

Term test for Sem 2, AY19/20.

- $e^x \ge x+1$ for all $x \in \mathbb{R}^+$, in particularly, $x \ge \log(x+1) > \log(x)$ **for all x positive**. This is particularly useful in proving $n\log(n)$ in $O(n^2)$, for example.
- (Q2) Trick for $\sum_{j=1}^{n} j \cdot 2^j$ and $\sum_{j=1}^{n} j/(2^j)$:

$$\begin{aligned} \sum_{j=1}^{n} j \cdot 2^j &= \sum_{j=1}^{n} \sum_{k=1}^{j} 2^j \\ &= \sum_{k=1}^{n} \sum_{j=k}^{n} 2^j \\ &= \sum_{k=1}^{n} 2^{n+1} - 2^k \\ &= n \cdot (2^{n+1}) - (2^{n+1} - 2) \\ &= \Theta(n \cdot 2^n). \end{aligned}$$

- Similarly,

$$\begin{aligned} \sum_{j=1}^{n} \frac{j}{2^j} &= \sum_{j=1}^{n} \sum_{k=1}^{j} \frac{1}{2^j} \\ &= \sum_{k=1}^{n} \sum_{j=k}^{n} \frac{1}{2^j} \\ &= \sum_{k=1}^{n} \left(\frac{1}{2^{k-1}} - \frac{1}{2^n}\right) \\ &= 2 - \frac{n}{2^n} = \Theta(1). \end{aligned}$$

- Note that some parts of the computation may be wrong; but I have checked that asymptotic bound is indeed correct.
- log tricks: $\frac{1}{\log_a(b)} = \log_b(a)$, "base transfer" $\frac{\log_c(b)}{\log_c(a)} = \log_a(b)$ (proof can be done using "chain rule" as follows: $\log_a(b) * \log_c(a) = \log_c(b)$, and then homogenize), $a^{\log_a(b)} = b$.
- $log(x!) = x\log(x)$ for any x. More specifically, **Stirling's** says that $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.
- This can be applied in weird shit like $\log((\log(\log(n)))!)$ which is equal to $\log(\log(n)) * \log(\log(\log(n)))$.
- $2^{\log(\log(\log(n)))}$ has different complexity with $10^{\log(\log(\log(n)))}$. Use the third and second log trick, and we can deduce that the first one is $\log(\log(n))$ and second is $\text{first}^{\log_2(10)}$.
- Similarly, 2^{2n} and 2^n has different complexity. Don't take constants in powers (especially "after taking log") for granted.
- Letting $n = 2^k$ does wonders to weird log stuff, and taking log of both sides does wonders to weird power stuff.
- For algorithm-constructing questions, make sure you **read the question carefully** and understand wholly what the algorithm does (*at all its phases/stages.*) This paper's Q3 is honestly quite confusing and tedious.

Term test for Sem 2, AY20/21. This test is very technical/math-heavy, so I won't put too much here.

- For Q3, remember to explicitly **state base cases** for runtime analysis,
- and then also remember to provide **a valid construction** for the "decision tree classes"; i.e. configurations must satisfy the property that "the peak must be in the specified location".

Term test for Sem 2, AY18/19.

- Insertion sort: we can always compare the functions iteratively, and "insert" them to an existing total-order-of-functions. This reduces the need to "compare all functions with one another".
- **Be careful with asymptotic constants.** For example, to prove $(1+\cos(n\pi/2)) = \Theta(1)$, need to set $c \ge 2$, as left function will get arbitrarily close to 2. Cannot just say "works for any c".
- Weird phrasing of Q4(b). Read question multiple times and understand what its "best meaning is"; in this case, it's *expected number of rounds where score is positive, given that* ***they play an infinite round of games***.

September 2008 Test.

- My standard tips for MCQ test: **Remember that MCQs is created to facilitate learning**. Pick answer that is **most related** to others, and **try to avoid** "None of the Above" answers unless everything else seems either trap-py, plain wrong, or unrelated.
- **Searching** for a particular thing means not assuming the thing is in there (based on Q12 of this paper's answer key).
- Preorder traversal is **not** the reverse order of postorder traversal. Preorder is parent, left, right and postorder is left, right, parent.

Sample Midterm Review. 2 more materials "unique" to this problem set: adversarial argument and hashing.

- **Adversarial** means creating 2 inputs that produces **same output** given an (arbitrary) "iteration" of an algorithm. Note that the second input can be "dependent" on how the algorithm behaves, as "the algorithm is not supposed to know everything".
- Namely, we can fix input I, and we can adjust an *adversary* I' based on how the algorithm acts (deterministically) on I. So, in other words, we prove that the "degree of freedom" of the algorithm is "not enough".

- In Q2's case, we craft a nice I and set "$2n-1$ degrees of freedom", and we set I' to be based on a "degree of freedom" that the algorithm has not "touched". In this case, it is a pair of consecutive numbers.
- Hashing lingo: choose h from a family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ (the keys/input values) to $[M = \Theta(n)]$ (the storages/memory/buckets).
- *Insert* to hash table.
- Collisions (of expected $\Theta(1)$ occurences) to be handled via chaining (no need fancy stuff such as quadratic probing).
- Correctness is assured, as for a given sampled hash function, *everything from there is deterministic*.
- ***Note:*** if U is so large that $\log(U)$ does not fit in a machine word, then (all) operations cannot be done in constant time. In practice, such situations are rare, as memory is typically big enough to store *pointers to input elements*.