Recitation 1: Asymptotic Notation, Recurrences, and Probability

1 Asymptotic Notation

1.1 Definitions

First let us review the definitions.

Definition 1 *"Big O": If $f(n) = O(g(n))$ then there exists $C > 0$ such that there exists $N > 0$ where for all $n > N$, $0 \leq f(n) \leq Cg(n)$.*

In English, $f(n) = O(g(n))$ if for very large n, $f(n)$ is smaller than $Cg(n)$ for some sufficiently large constant C.

Definition 2 *"Little o": If $f(n) = o(g(n))$ then for all $\epsilon > 0$, there exists N where for all $n > N$, $0 \leq f(n) < \epsilon g(n)$.*

In English, $f(n) = o(g(n))$ if for very large n, $f(n)$ becomes arbitrarily smaller than $g(n)$.

Definition 3 *"Big Omega": If $f(n) = \Omega(g(n))$ then there exists $C > 0$ such that there exists $N > 0$ where for all $n > N$, $f(n) \geq Cg(n) \geq 0$.*

In English, $f(n) = \Omega(g(n))$ if for very large n, $f(n)$ is larger than $Cg(n)$ for some sufficiently large constant C.

Definition 4 *"Little omega": If $f(n) = \omega(g(n))$ then for all $\epsilon > 0$ there exists N where for all $n > N$, $f(n) > \epsilon g(n) \geq 0$.*

In English, $f(n) = \omega(g(n))$ if for very large n, $f(n)$ becomes arbitrarily larger than $g(n)$.

Definition 5 *"Theta": If $f(n) = \Theta(g(n))$ then there exists $a, b > 0$ such that there exists N where for all $n > N$, $0 \leq af(n) \leq g(n) \leq bf(n)$.*

In English, $f(n) = \Theta(g(n))$ if for very large n, $f(n)$ and $g(n)$ grow at the same speed up to a constant multiple.

Intuitively, we are ignoring constant factors and low-order terms to focus only on the asymptotic order of growth of the functions. A useful mnemonic device may be to think of o as meaning strictly less than, O as smaller than or equal to, Ω as greater than or equal to, and ω as strictly greater than, and Θ as equal to.

This mneumonic device is helpful, but in what cases does the analogy break down? When working with numbers, given two values a and b where $a \leq b$, then it must be the case that either $a < b$ or $a = b$. Does this hold for functions? What is an example of two functions f and g, such that $f(n) = O(g(n))$, but $f(n) \neq o(g(n))$ and $f(n) \neq \Theta(g(n))$? How do we know which of these forms to use when describing an algorithm?

1.2 Examples

1. $f(n) = 5$: is $O(n^2)$, and $o(n^2)$; is NOT $\Theta(n^2)$ or $\Omega(n^2)$.
2. $f(n) = 3n^3 + n^2$: is $O(n^3)$, $\Omega(n^3)$, and $\Theta(n^3)$; is NOT $o(n^3)$.
3. $f(n) = n^{0.000001}$: is $\Omega(\lg(n))$; is NOT $\Theta(\lg n)$, $O(\lg n)$, or $o(\lg n)$.
4. $f(n) = n + \sin(n)$: is $O(n)$, $\Omega(n)$, and $\Theta(n)$.

1.3 Practice Problems

Consider the following functions. Within each group, sort the functions in asymptotically increasing order, showing strict orderings as necessary. For example, we may sort $n^3, n, 2n$ as $2n = O(n) = o(n^3)$.

1. $\log n$, $\sqrt[3]{n}$, $\log_7 n$, $\log n^3$, $\log\log n^{10}$, $\log^3 n$.

   Solution: $\log\log n^{10} = o(\log n) = O(\log n^3) = O(\log_7 n) = o(\log^3 n) = o(\sqrt[3]{n})$.

   Note that a change of base in the logarithm is equivalent to a constant factor, so they are asymptotically equal.
   A change of variables can be helpful here - if you're not sure whether $\log^3 n$ is asymptotically bigger or smaller than $O(\sqrt[3]{n})$, replacing n with 2^m and comparing m^3 with $2^{m/3}$ is easier. Note also that $\log n^3 = 3\log n$. Also note that $\log\log n^{10} = \log(10\log n) = \log\log n + O(1)$.

2. $n^{5/3}$, $n\log n^2$, $n^2\log n$, 2^n, $\log(n!)$.

   Solution: $\log(n!) = O(n\log n^2) = o(n^{\frac{5}{3}}) = o(n^2\log n) = o(2^n)$.

   Here, we use Stirling's approximation for the factorial which gives $\log n! = O(n\log n)$.

3. $(\log n)^n$, $(\log n)^{n-5}$, e^n, $n^{10\log n}$.

   Solution: $n^{10\log n} = o(e^n) = o((\log n)^{n-5}) = o((\log n)^n)$.

Note that $(\log n)^{n-5}$ and $(\log n)^n$ are asymptotically different because the constant is in the exponent. To compare e^n and $(\log n)^{n-5}$, replace n with 2^m. Then, we see that e^{2^m} is asymptotically smaller than $m^{(2^m-5)}$. To compare $n^{10\log n}$ and e^n, take the log of each to get $10\log^2 n$ and $n\log e$. Clearly, e^n is asymptotically larger.

2 Run time Recurrences

With many types of algorithms we want to solve the larger problem by splitting it into smaller parts, solving each of those and then merging the results. Solving algorithms like this often results in fast run times.

It can help from a design perspective because you only need to verify the correctness of the smallest problem and the correctness of the merging of two solved problems (CS is all about being lazy).

It can help from an analysis perspective because this method is so common, time analysis techniques and machinery have already been produced (woo double lazy!).

So what do these recurrences look like? Let $T(n)$ be the running time for the algorithm on a problem of size n. Standard divide-and-conquer algorithms often result in a recurrence of the form

$$T(n) = aT(n/b) + f(n).$$

where $a \geq 1, b > 1$.

2.1 Master Theorem

The master theorem is a general method to analyze recurrence relations of the above form. There are three cases:

1. If $f(n)$ polynomially less than $n^{\log_b(a)}$ (i.e. $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$), then $T(n) = \Theta(n^{\log_b(a)})$.
2. If $f(n) = \Theta(n^{\log_b(a)}\log^k(n))$ for some constant $k \geq 0$, then $T(n) = \Theta(f(n)\log(n)) = \Theta(n^{\log_b(a)}\log^{k+1}(n))$.
3. If $f(n)$ is polynomially greater than $n^{\log_b(a)}$ (i.e. $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$), and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

If $n^{\log_b(a)}$ is greater, but not polynomially greater, than $f(n)$, the Master Theorem cannot be used to determine a precise bound. An example of this is the recurrence $T(n) = 2T(n/2) + \Theta(n/\log(n))$. (The solution to this recurrence is $T(n) = O(n\log\log n)$, which

can be arrived at by unrolling the recurrence and using the fact that the harmonic series is $O(\log n)$.)

The Master theorem essentially compares the weight at the root of the recurrence tree, $f(n)$, with the number of leaves in the tree, $n^{\log_b a}$.

2.2 Substitution

Substitution of variables is often helpful in solving recurrences. For instance, suppose we wish to solve the recurrence $T(n) = 2T(\sqrt{n}) + 1$, which we've already seen does not fall into the scope of the Master theorem. Let us create a new variable m such that $n = 2^m$. We now have

$$T(2^m) = 2T(2^{m/2}) + 1.$$

Furthermore, we can create a new function $S(m) = T(2^m)$, so that

$$S(m) = 2S(m/2) + 1.$$

Notice that $S(m)$ is in a form where we can apply the Master theorem! Therefore, we have $S(m) = O(m)$, and $T(n) = O(m) = O(\log n)$.

2.3 Guess and Check

You can often determine the solution for a recurrence by inspection. In that case, you can just guess the solution to the recurrence and then prove that it is correct by induction. Some examples include:

1. $T(n) = 2T(n/4) + n$ | ($T(n) = O(n)$)
2. $T(n) = 2T(n/2) + 1$ | ($T(n) = O(n)$)
3. $T(n) = 4T(n/2) + n$ | ($T(n) = O(n^2)$)
4. $T(n) = 2T(n/2) + n$ | ($T(n) = O(n \log n)$, e.g. mergesort). Here it's crucial that guess-and-check is done with $T(n) \leq n \log_2 n$, and using the fact that $\log_2 \frac{n}{2} = \log_2 n - 1$.

A common mistake: When you want to guess, you have to come up with the closed-form expression and not just asymptotic notation. For example, consider $T(n) = 2T(n/4) + 1$. Suppose you guess that $T(n) = O(1)$. It seems it is correct, since $T(n) = 4O(1) + 1 = O(1)$. However, this is not true. If we guessed a closed-from expression of a function, namely c for a constant c, we would not be able to show that $T(n) = O(1)$. Let's try to prove $T(n) \leq c$. By induction hypothesis, assume $T(m) \leq c$ for any $m < n$, then we can only show that $T(n) \leq 2c + 1$. This does not imply that $T(n)$ is also at most c. Thus, the induction fails.

More subtly, note that a function failing to satisfy a recurrence does not on its own indicate whether the actual solution to the recurrence is large or smaller. As a contrived example, consider the recurrence $T(n) = 2T(n/2)+1$. The guess $T(n) = cn+1$ fails, as $2T(n/2)+1 = 2(cn/2+1)+1 = cn+3 \not\leq cn+1$.

2.4 Example: Mergesort

As an example, we will use Mergesort and analyze its running time by defining a recurrence.

Mergesort divides the problem in two halves, solves the two halves recursively and it merges them by iterating through all the elements in the two sublists.

Let $T(n)$ be defined as the number of operations required to sort a list of n elements using mergesort. Then, $T(n)$ is equal to the time to solve the two subproblems plus the time to merge the two sublists. Since the two lists are of length $n/2$, and since merging iterates through all the n elements, we know that:

$$T(n) = 2T(n/2) + \Theta(n) \tag{1}$$

We solve this recurrence by using the Master Theorem. Using the notation from the previous part, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$. Since $f(n)$ is $\Theta(n)$, we are in the second case of the Master Theorem. Consequently, we can apply the theorem to conclude that $T(n) = \Theta(n \log n)$.

2.5 Practice Problems

Let $T(n)$ be the time complexity of an algorithm to solve a problem of size n. Assume $T(n)$ is $O(1)$ for any $n < 3$. Solve the following recurrence relations for $T(n)$.

1. $T(n) = 5T\left(\frac{n}{2}\right) + n^3$.

   Solution: By case 3 of the Master Theorem, since $f(n) = n^3$ is polynomially greater than $n^{\log_2 5}$ and $5(\frac{n}{2})^3 \leq cn^3$ for $c = \frac{5}{8} < 1$, $T(n) = \Theta(f(n)) = \Theta(n^3)$.

2. $T(n) = 3T\left(\frac{n}{3}\right) + n$.

   Solution: By case 2 of the Master Theorem, since $f(n) = n$ is $\Theta(n^{\log_3 3})$, $T(n) = \Theta(n \log n)$.

3. $T(n) = 5T(\sqrt[2]{n}) + \log\log n$.

Solution: We solve this one by changing the variables. Assume that n is a power of 2, and let $n = 2^m$, then

$$T(2^m) = 5T(2^{m/2}) + \log m.$$

If we define $S(m) = T(2^m)$, the recurrence becomes

$$S(m) = 5S(m/2) + \log m.$$

We can use the Master Theorem (case 1), since $m^{\log_2 5}$ is polynomially greater than $\log m$. Therefore, $S(m) = \Theta(m^{\log_2 5})$, and $T(2^m) = \Theta(m^{\log_2 5})$. Changing the variable back ($m = \log n$), we have $T(n) = \Theta(\log^{\log_2 5} n)$.

4. $T(n) = 3T(n/5) + 2T(n/7) + \Theta(n)$.

   Solution: We can do this in two ways:

   (a) We can use intuition, given that $3n/5 + 2n/7 < n$, and guess a solution of the form $T(n) \leq cn$ for some $c > 0$. We can prove this using strong induction on n. Assume $T(n) \leq cn$ for all $n < k$ for some k. We will show that $T(k) \leq ck$.

   $$T(k) = 3T(k/5) + 2T(k/7) + c_2 k$$

   for some fixed c_2. Now by our inductive hypothesis,

   $$\begin{aligned} T(k) &\leq \frac{3}{5}ck + \frac{2}{7}ck + c_2 k \\ &= \left(\frac{31}{35}c + c_2\right) k \leq ck \end{aligned}$$

   whenever $c \geq \frac{35}{4}c_2$. In addition, for our base case $k = 1$ to hold, we pick $c = \max\left(\frac{35}{4}c_2, T(1)\right)$. This proves our claim for large enough c, by induction.

   (b) A more complicated calculation can be done with the help of a recursion tree. Let $d > 0$ represent the constant factor in the term $\Theta(n)$. We use the following claim which proof we defer.

   Claim: The sum of the sizes of all sub-problems at the k-th level is at most $(31/35)^k n$.

   Assuming the claim and by the linearity of the cost term (the cost of problems of size m is dm), the total cost of nodes at level k is at most $\left(\frac{31}{35}\right)^k dn$. The sum of the costs at all levels is $\sum(\frac{31}{35})^k dn$, which is a geometric sum converging to $\Theta(n)$.

   We now prove the claim by induction on k. This clearly holds for the root of the tree ($k = 0$). Let m_i for $i = 1, ..., K$, be the sub-problem sizes of nodes at

level k and note that the actual value of K is not important. By the induction hypothesis, $\sum m_i \leq \left(\frac{31}{35}\right)^k n$. Each sub-problem of size m_i generates 3 sub-problems at level $k+1$ of size $m_i/5$ and 2 sub-problems of size $m_i/7$. Therefore, the total size fo sub-problems at level $k+1$ is

$$\begin{aligned}\sum_i \left(\frac{3}{5}m_i + \frac{2}{7}m_i\right) &= \sum_i \left(\frac{3}{5} + \frac{2}{7}\right) m_i \\ &= \left(\frac{31}{35}\right) \sum_i m_i \\ &\leq \left(\frac{31}{35}\right)^{k+1} n.\end{aligned}$$

3 Probability Practice

This section mostly contains practice problems. See the probability review handout for further notes and explanation.

3.1 Expectation and Indicator Variables

1. There is a dinner party where n people check their hats. The hats are mixed up during dinner, so that afterward each person receives a random hat. In particular, each person gets their own hat with probability $\frac{1}{n}$. What is the expected number of people who get their own hat?

   Solution: Letting G be the number of people that get their own hat, we want to find $E[G]$.

   The trick is to express G as a sum of indicator variables. In particular, let G_i be an indicator for the event that the i^{th} person gets their own hat. That is, $G_i = 1$ if they get their own hat, and $G_i = 0$ otherwise. The number of people that get their own hat is the sum of these indicators: $G = \sum_{i=1}^{n} G_i$.

   Note that these indicator variables are not mutually independent. For example, if $n-1$ people all get their own hats, then the last person is certain to receive their own hat. But, since we plan to use linearity of expectation, we don't have worry about independence!

   Now since G_i is an indicator, we know $E[G_i] = \Pr[G_i = 1] = \frac{1}{n}$. Applying linearity

of expectation gives us:

$$\begin{aligned} E[G] &= \sum_{i=1}^{n} E[G_i] \\ &= \sum_{i=1}^{n} \frac{1}{n} = \boxed{1}. \end{aligned}$$

So even though we don't know much about how hats are scrambled, we've figured out that on average, just one person gets their own hat back!

2. Alice, Bob, Chris, and Daisy were all born in 2019. What is the expected value of the number of distinct birthdays the four of them have?

Solution: Let X_i be the indicator variable for whether or not someone has a birthday on day i for $1 \leq i \leq 365$. We want to compute $E[X]$ where $X = \sum_{i=1}^{365} X_i$ is the number of distinct birthdays. By linearity of expectation, we have

$$E[X] = \sum_{i=1}^{365} E[X_i].$$

Now to compute $E[X_i]$, we need to compute the probability that someone has a birthday on day i.

To do this, we compute the probability that no one has a birthday on day i, which is $\left(\frac{364}{365}\right)^4$. So $\Pr[X_i] = 1 - \left(\frac{364}{365}\right)^4$. Plugging this in gives us,

$$\begin{aligned} E[X] &= \sum_{i=1}^{365} E[X_i] \\ &= \sum_{i=1}^{365} \Pr[X_i] \\ &= \sum_{i=1}^{365} 1 - \left(\frac{364}{365}\right)^4 \\ &= \boxed{365\left(1 - \left(\frac{364}{365}\right)^4\right)} \end{aligned}$$

3.2 Probability Bounds

Find an upper bound on the probability of flipping more than $\frac{3}{4}n$ heads in a sequence of n coin tosses.

We can either use Markov's inequality, Chebyshev's inequality, or Chernoff bounds for this question. We will demonstrate how to use all three for this question.. They are listed below as a reminder.

Markov's Inequality

Let Y be a discrete random variable taking non-negative values in the set S. Then for any $a > 0$,

$$\Pr[Y \geq a] \leq \frac{E[Y]}{a}$$

Chebyshev's Inequality

Let X be a random variable with expected value μ and strictly positive variance σ^2. Then for all real $k > 0$:

$$\Pr[|X - \mu| \geq k] \leq \frac{\sigma^2}{k^2}$$

Chernoff Bounds

Suppose $X_1, \ldots, X_n$ are independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $\mu = \mathbb{E}[X]$ denote the sum's expected value. Then for any $\beta > 0$,

- $\Pr[X > (1+\beta)\mu] < e^{-\beta^2\mu/3}$, for $0 < \beta < 1$
- $\Pr[X > (1+\beta)\mu] < e^{-\beta\mu/3}$, for $\beta > 1$
- $\Pr[X < (1-\beta)\mu] < e^{-\beta^2\mu/2}$, for $0 < \beta < 1$

1. *Find the expected value.* For all of these bounds, you first need to find the expected value of the quantity you are trying to bound. Let $Y_1, Y_2, \ldots, Y_n$ be independent outcomes of fair coin where

$$Y_i = \begin{cases} 0 & \text{if } i^{th} \text{ toss is heads} \\ 1 & \text{otherwise} \end{cases}$$

   and $Y = \sum_{i=1}^{n} Y_i$ is the total number of heads in the sequence.

   By linearity of expectation, we have $E[Y] = \sum_{i=1}^{n} E[Y_i] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$.

2. *Pick a concentration inequality to bound the desired probability.* Usually, you will need to pick a bound depending on what the problem is asking. Here, we will show how to use all three of the ones given in the probability review handout.

3.2.1 Markov's Inequality

We can use Markov's inequality since Y is a non-negative random variable. This gives us

$$\Pr\left[Y \geq \frac{3}{4}n\right] \leq \frac{E[Y]}{\frac{3}{4}n} = \frac{\frac{1}{2}n}{\frac{3}{4}n} = \frac{2}{3}.$$

This is a decent upper bound, and if this is sufficient to solve the problem, you should use Markov, since it is the easiest to apply. But if we need a stronger bound, we can apply Chebyshev or Chernoff.

3.2.2 Chebyshev's Inequality

To apply Chebyshev, we need to compute the variance of Y. This obviously requires more work than applying Markov's inequality, so again, if Markov is sufficient to solve the problem, use Markov. You should only apply Chebyshev if you need a stronger bound.

To compute $Var[Y]$, we can use the known variance for a binomial distribution or re-compute it as follows. Since all the Y_i are independent of one another, we have

$$Var[Y] = \sum_{i=1}^{n} Var[Y_i].$$

We have

$$\begin{aligned} Var[Y_i] &= E[Y_i^2] - E[Y_i]^2 \\ &= \frac{1}{2} - \frac{1}{4} = \frac{1}{4}. \end{aligned}$$

So finally, we have

$$Var[Y] = \sum_{i=1}^{n} \frac{1}{4} = \frac{n}{4}.$$

Plugging this into Chebyshev's inequality gives:

$$\Pr\left[\left|Y - \frac{n}{2}\right| \geq \frac{n}{4}\right] \leq \frac{n/4}{n^2/4^2} = \frac{4}{n}.$$

This finally gives us

$$\Pr\left[Y \geq \frac{3}{4}n\right] \leq \Pr\left[\left|Y - \frac{n}{2}\right| \geq \frac{n}{4}\right] \leq \frac{n/4}{n^2/4^2} = \frac{4}{n}.$$

This is a better bound than Markov, that gets arbitrarily small as n goes to infinity. Sometimes, however, we will want a bound that gets exponentially small as n goes to infinity. In this situation, we would use Chernoff.

3.2.3 Chernoff Bound

To use Chernoff, you MUST have all of your indicator variables be mutually independent. Luckily, in this case all the Y_i are independent of one another, so we can apply Chernoff.

In order to get the probability we desire, we choose $\beta = \frac{1}{2}$ to obtain

$$\begin{aligned}\Pr\left[Y > \frac{3}{4}n\right] &= \Pr\left[Y > \left(1+\frac{1}{2}\right)E[Y]\right] \\ &\leq e^{-\frac{1}{4}\cdot\frac{n}{2}\cdot\frac{1}{3}} \\ &= e^{-n/24}.\end{aligned}$$

This is by far, the strongest bound, as long as all the conditions to be able to use Chernoff are met.

3.3 Other Probability Bounding Techniques

3.3.1 Repeated Trials

Suppose we have a randomized algorithm $\mathcal{A}$ that returns an n-bit prime with probability $\frac{1}{50}$ and returns nothing otherwise. Design an algorithm that finds an n-bit prime with probability at least $\frac{2}{3}$.

Solution: Run $\mathcal{A}$ $\lceil \log_{49/50}(1/3) \rceil = 55$ times and return the first prime that is found. Our new algorithm finds a prime if at least one run of $\mathcal{A}$ succeeds. The probability that no runs of $\mathcal{A}$ succeed is $\left(\frac{49}{50}\right)^{55} < \left(\frac{49}{50}\right)^{\log_{49/50}(1/3)} = \frac{1}{3}$. Thus, our algorithm succeeds with probability at least $\frac{2}{3}$.

3.3.2 Union Bound

Union Bound
Consider events $A_1, \ldots, A_n \subseteq \Omega$, where Ω is a sample space. Then we have

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \Pr\left[A_i\right]$$

Suppose we have a randomized algorithm $\mathcal{A}$ that assigns n processors to $3n$ jobs. Suppose we also know that $\mathcal{A}$ assigns each processor i at most 5 jobs with probability at least $1 - \frac{1}{n^2}$. Find a lower bound on the probability that all processors are assigned at most 5 jobs.

Solution: Let P_i be the event that processor i gets assigned more than 5 jobs, and let $P = \bigcup_{i=1}^{n} P_i$ be the event that at least one processor gets assigned more than 5 jobs. We know $\Pr[P_i] \leq \frac{1}{n^2}$. By Union Bound,

$$\Pr[P] \leq \sum_{i=1}^{n} \Pr[P_i] \leq \sum_{i=1}^{n} \frac{1}{n^2} = n \cdot \frac{1}{n^2} = \frac{1}{n}.$$

Since the probability we are looking for is the opposite of P (that all processors are assigned at most 5 jobs), our lower bound is $1 - \frac{1}{n}$.

Common mistake: Using an approach similar to repeated trials would be incorrect here. We could have said something like the probability that all n processors are assigned at most 5 jobs is at least $\left(1 - \frac{1}{n^2}\right)^n$, multiplying all the n probabilities. However, you cannot multiply probabilities *unless they are independent*. The number of jobs each processor gets assigned are not independent from one another. In the repeated trials example, the randomness in each run of the algorithm is independent from the randomness in other runs of the algorithm.

4 Randomized Algorithms

A randomized algorithm is an algorithm that is allowed to use randomly generated numbers to make decisions. This means, given the same input, the same randomized algorithm can have:

1. different runtimes
2. different outputs.

A **Monte Carlo** algorithm always runs in *polynomial time* and gives the *correct output with high probability*.

A **Las Vegas** algorithm runs in *expected polynomial time* and always gives the *correct output*.

Exercise 1: Given a Monte Carlo algorithm $\mathcal{A}$ and a polynomial-time algorithm $\mathcal{V}$ which can verify whether $\mathcal{A}$ returned a correct answer, devise a Las Vegas algorithm that performs the same task using the repeated trials technique above.

Exercise 2: Given a Las Vegas algorithm $\mathcal{B}$, devise a Monte Carlo algorithm that performs the same task, and prove that it returns the correct output with high probability using a probability bounding technique above.

5 Appendix: Recursion Trees

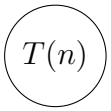
A recursion tree can help to visualize where the algorithm is doing the most work– at the root of the tree, at the leaves, or spread evenly throughout (or perhaps something more complicated).

Basic idea: draw a node for each instance of the problem, and draw edges to represent the parent-child relationships among the problems. In each node, write the cost incurred directly at that subproblem ($f(n)$ for the particular n of the subproblem). The total cost of the algorithm may be computed by summing up the costs of all the nodes.

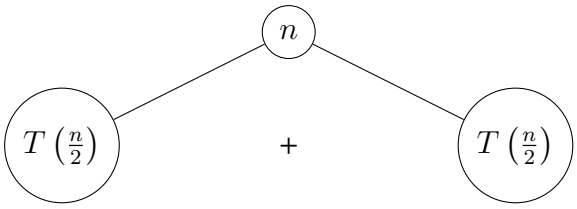
The branching factor for the tree is a. The height (number of levels) is $\log_b(n)$. At a given level h levels below the root, each node incurs $f(\frac{n}{b^h})$. At a given level h levels below the root, there are a^h nodes. Summing across the tree, the cost of a given level h levels below the root will be $f(\frac{n}{b^h})a^h$.

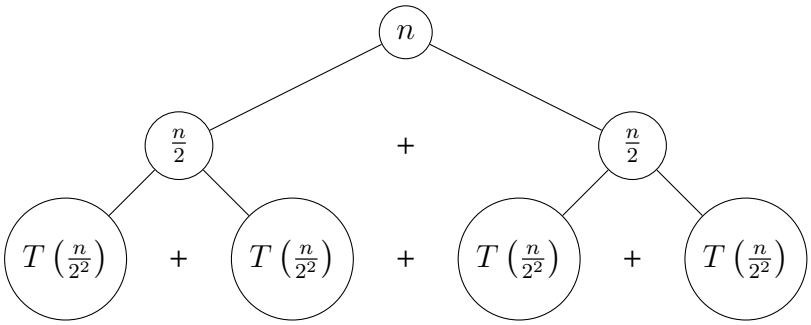
5.0.1 Building a recursion tree for $T(n) = 2T(n/2) + n$

- Level 0 :



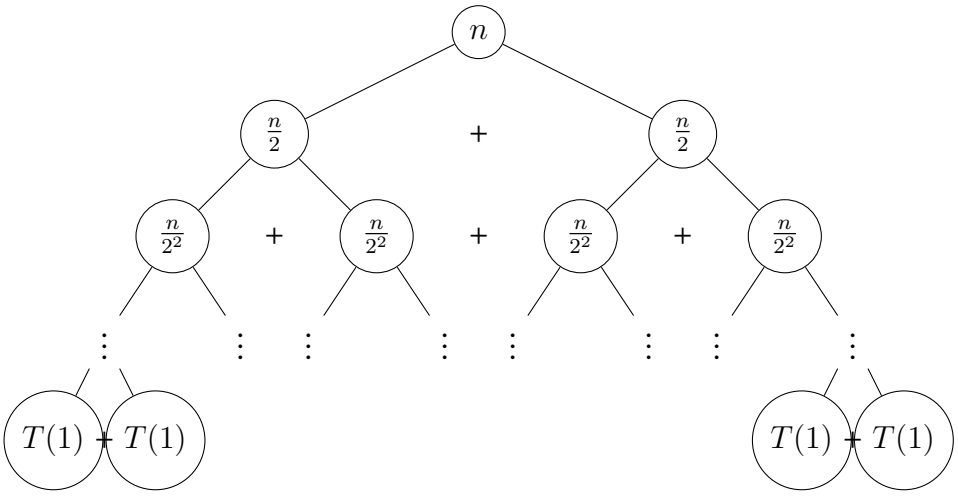
- Level 1:



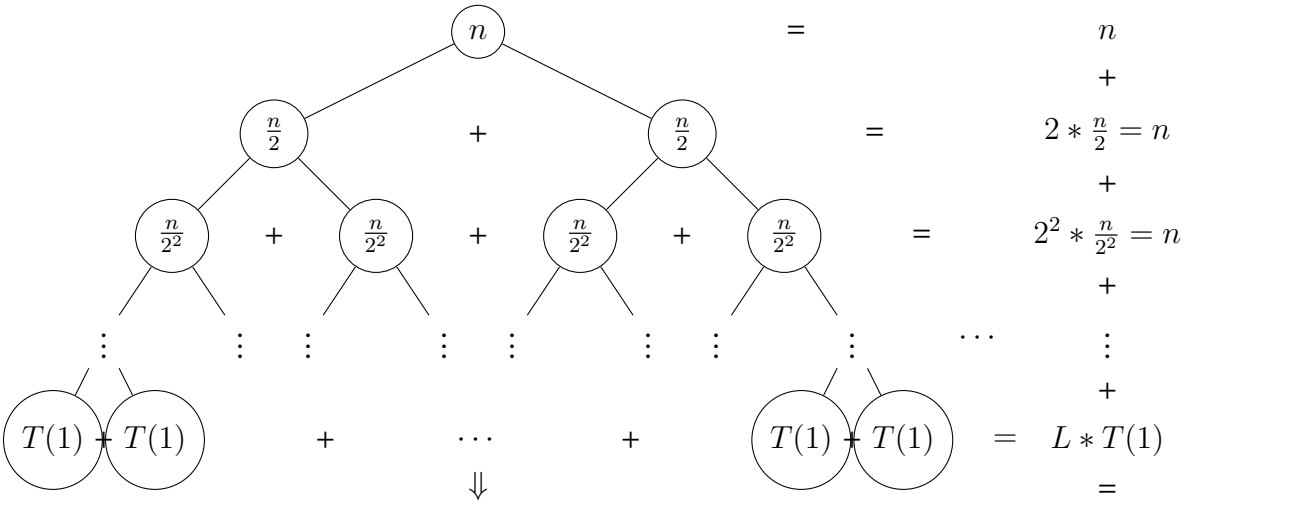


- Level 2:

  ⋮



- Level $k = \log_2 n$:



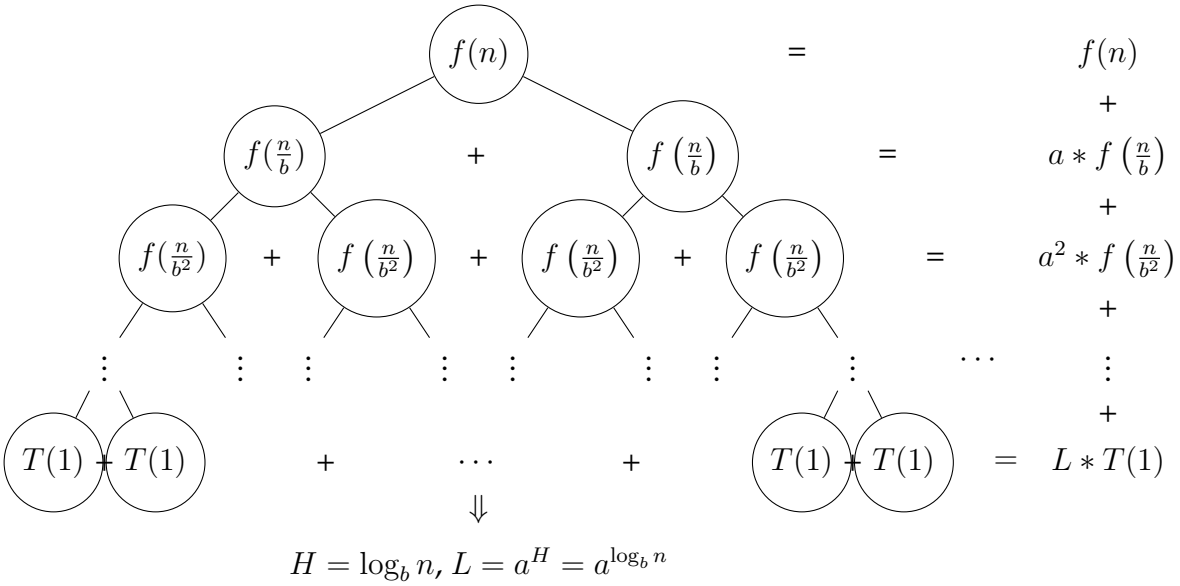
$$H = \log_2 n,\ L = 2^H = 2^{\log_2 n} = n \qquad \Leftrightarrow \quad n + n * \log_2 n = O\left(n \cdot \log n\right)$$

Why does $n^{\log_b(a)}$ shows up a lot in recurrences?

It shows up because this is a count of the number of leaves of the recursion tree. The

number of leaves is a^H where H is the height of the tree, $\log_b n$. We then have $a^H = a^{\log_b(n)} = n^{\log_b(a)}$.

Building a recursion tree for $T(n) = aT(n/b) + f(n)$



The analysis of the overall running time of the general case depends on the comparison of $f(n)$ and $n^{\log_b a}$. The proof of the exact analysis of the running time to the general case is what is known as the "Master Theorem". It is a proof by cases that can be found in Chapter 4.6 of CLRS.