CSE 101 - ALGORITHMS - SUMMER 2000
Lecture Notes 2
Wednesday, July 5, 2000.

Chapter 2

Mathematical Foundations

This chapter presents the fundamental mathematical notions, notations and formulae needed to analyze the running time of simple algorithms. We assume the reader familiar with well-known properties of logarithms and summations, such as

- $\log_a(b \cdot c) = \log_a(b) + \log_a(c)$,
- $\log_a(b/c) = \log_a(b) - \log_a(c)$,
- $\log_a(b^c) = c \cdot \log_a(b)$,
- $c^{\log_a(b)} = b^{\log_a(c)}$,
- $1 + x + x^2 + \cdots + x^n = \dfrac{x^{n+1} - 1}{x - 1}$,

as well as with basic concepts of set theory and with simple mathematical reasoning and calculus.

The functions considered in this chapter and most of the rest of the course take natural numbers as arguments. When not otherwise specified, a function has only one argument and that argument is a natural number.

2.1 Growth of Functions

The order of growth of a running time of an algorithm, as function of input size, gives a simple and usualy appropriate characterization of the efficiency of an algorithm. We saw that the running time of merge sort was $\Theta(n \log n)$ while that of insertion sort was $\Theta(n^2)$, meaning that merge sort is faster than insertion sort on inputs of large enough size. It is worth noticing that in practice, insertion sort may be more efficient than merge sort on many inputs. In this course, we only study *asymptotic* efficiency of algorithms, that is, we are only concerned with how the running time of an algorithm increases as the size of the input increases toward infinity.

This section introduces some mathematical notation which significantly simplifies the asymptotic analysis of algorithms.

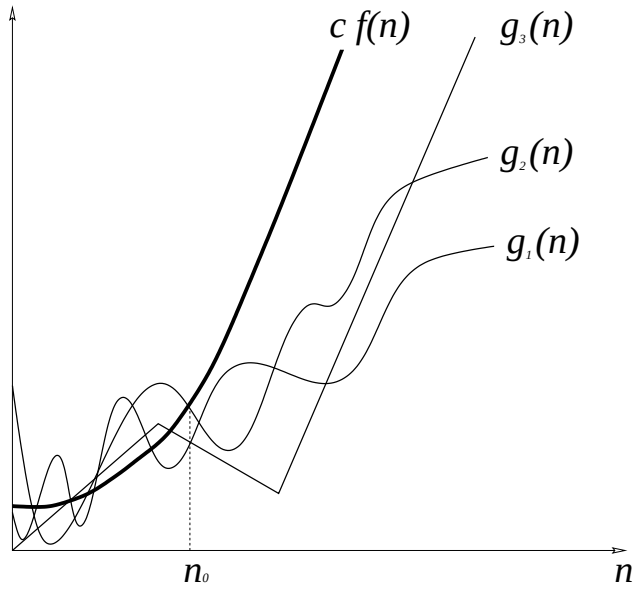
2.1.1 O-Notation

Given a function f, we let $O(f(n))$ denote the class of functions g for which f is an *asymptotically upper bound*. Formally,

$$O(f(n)) = \{g \mid (\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\ 0 \leq g(n) \leq cf(n)\}$$

that is, $O(f(n))$ is the set of functions g for which there are some constants $c > 0$ and $n_0 \geq 0$ such that $g(n)$ is positive and smaller than $cf(n)$ for each n larger than n_0. Notice that for each c and n_0 as above, one can always choose any larger c' and n_0' with the same property. In particular, given a finite number of functions $g_1, g_2, ..., g_k$ in

$O(f(n))$, one can choose a constant c and a number n_0 such that $g_i(n)$ is positive and smaller than $cf(n)$ for for all $n \geq n_0$ and $1 \leq i \leq k$. The following picture shows three functions in $O(f(n))$:



If the running time $T(n)$ of an algorithm $\mathcal{A}$ can be shown to be in $O(f(n))$ for some function f, then we say that $\mathcal{A}$ is "more efficient" than any algorithm running in time $f(n)$, even if this may not be true for all the inputs. However, it is true for inputs of large enough sizes. In fact, it would be more appropriate to say "asymptotically more efficient".

Exercise 2.1 Show that $10^{10}n^2 + 2^{100}n$ and $10^{-10}n^2 - 2^{-100}n$ are in $O(n^2)$.

Exercise 2.2 Show that $n \log n \in O(n^2)$.

Exercise 2.3 If $g(n) \leq f(n)$ for all $n \geq 0$, then $g \in O(f(n))$.

Exercise 2.4 Given $g \in O(f(n))$, show that $f + g \in O(f(n))$ and that $f \cdot g \in O((f(n))^2)$.

Exercise 2.5 If $g \in O(f(n))$ then $O(g(n)) \subseteq O(f(n))$.

Exercise 2.6 Are there any different functions $f \neq g$ such that $g \in O(f(n))$ and $f \in O(g(n))$?

Exercise 2.7 If $g_1 \in O(f_1(n))$ and $g_2 \in O(f_2(n))$ then is it the case that $g_1 + g_2 \in O(f_1(n) + f_2(n))$ and $g_1 \cdot g_2 \in O(f_1(n) \cdot f_2(n))$? How about $g_1/g_2 \in O(f_1(n)/f_2(n))$? How about ${g_1}^{g_2} \in O(f_1(n)^{f_2(n)})$?

Exercise 2.8 Is it the case that for any functions f and g, either $g \in O(f(n))$ or $f \in O(g(n))$?

Proof: We give a counter-example: let $f(n) = 1 + (-1)^n$ and $g(n) = 1 - (-1)^n$ for all $n \geq 0$, so f takes the values $2, 0, 2, 0, 2, 0, ...$ and g takes the values $0, 2, 0, 2, 0, 2, ...$ For the sake of contradiction, suppose that $f \in O(g(n))$. Then there are $c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. But that means that $f(2k) \leq cg(2k)$ for all $k > n_0/2$, that is, that $2 \leq 0$. Contradiction. It can be similarly shown that $g \notin O(f(n))$.

Exercise 2.9 Is it the case that $f \cdot g \in O((f(n))^2) \cup O((g(n))^2)$ for any functions f and g ?

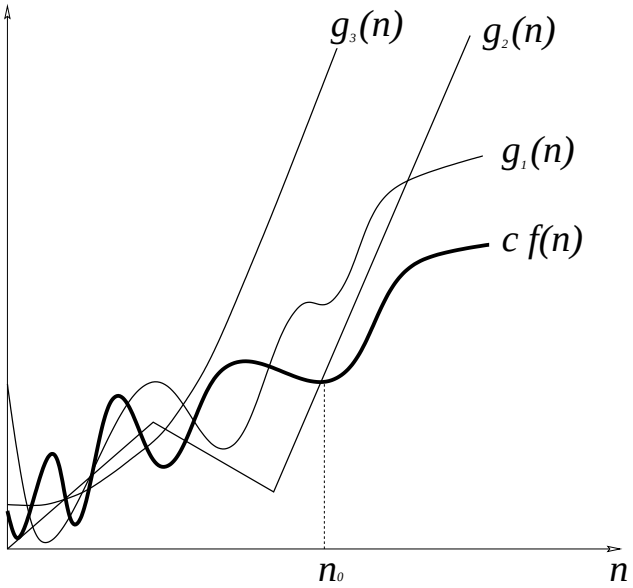
2.1.2 Ω-Notation

Given a function f, let $\Omega(f(n))$ denote the set of functions g for which f is an *asymptotically lower bound*, that is,

$$\Omega(f(n)) = \{g \mid (\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\ 0 \leq cf(n) \leq g(n)\}.$$

With other words, $\Omega(f(n))$ is the set of functions g for which there are some constants $c > 0$ and $n_0 \geq 0$ such that $g(n)$ is bigger than $cf(n)$ for each n larger than n_0. Notice also that for each c and n_0 as above, one can always choose

any larger c' and n_0' with the same property, so given a finite number of functions $g_1, g_2, ..., g_k$ in $O(f(n))$, one can choose a constant c and a number n_0 such that $g_i(n)$ is bigger than $cf(n)$ for for all $n \geq n_0$ and $1 \leq i \leq k$. The following picture shows three functions in $\Omega(f(n))$:



If the running time of an algorithm $\mathcal{A}$ is in $O(f(n))$ for some function f, then we say that $\mathcal{A}$ is "less efficient" than an algorithm running in time $f(n)$, even if this may not be true for all the inputs. As before, it is true for inputs of large enough sizes, so it would be more appropriate to say "asymptotically less efficient". The following proposition shows a natural immediate relationship between the Ω and the O notations:

Proposition 2.1 *Given functions f and g, then $g \in \Omega(f(n))$ if and only if $f \in O(g(n))$.*

Proof: *Exercise.*

Therefore, the Ω-notation is in some sense inverse to the O-notation. Prove the following exercises directly, without using Proposition 2.1.

Exercise 2.10 Show that the identity function $g(n) = n$ is both in $\Omega(\log n)$ and in $\Omega((\log n)^2)$.

Exercise 2.11 Show that 2^n is in $\Omega(n^{100})$.

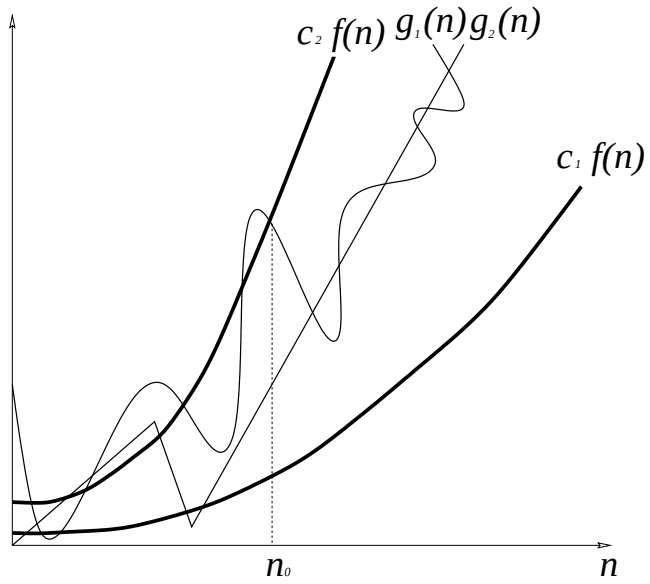
2.1.3 Θ-Notation

We now combine the O and Ω-notations. Given a function f, let $\Theta(f(n))$ denote the set of functions g for which f is an *asymptotically tight bound*, that is,

$$\Theta(f(n)) = \{g \mid (\exists 0 < c_1 < c_2)(\exists n_0 \geq 0)(\forall n \geq n_0)\ 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)\}.$$

Therefore, $\Theta(f(n))$ is the set of functions g for which there are some constants $0 < c_1 < c_2$ and $n_0 \geq 0$ such that $g(n)$ is bigger than $c_1 f(n)$ but smaller than $c_2 f(n)$ for each n larger than n_0. Notice again that for each c_1, c_2 and n_0 as above, one can choose any constants c_1', c_2' with $0 < c_1' < c_1 < c_2 < c_2'$ and any $n_0' > n_0$ such that the property in the definition of the Θ-notation also holds. Hence, given a finite number of functions $g_1, g_2, ..., g_k$ in $\Theta(f(n))$, one can choose constants c_1, c_2 and a number n_0 such that $g_i(n)$ is bigger than $c_1 f(n)$ and smaller than $c_2 f(n)$ for for all

$n \geq n_0$ and $1 \leq i \leq k$. The following picture shows two functions in $\Omega(f(n))$:



Proposition 2.2 $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$ *for any function* f.

Proof: *Exercise.*

Exercise 2.12 $\lceil f \rceil$ and $\lfloor f \rfloor$ are in $\Theta(f(n))$ for any function f, where $\lceil f \rceil$ and $\lfloor f \rfloor$ are the functions defined by $\lceil f \rceil(n) = \lceil f(n) \rceil$ and $\lfloor f \rfloor(n) = \lfloor f(n) \rfloor$, respectively.

Exercise 2.13 If $g \in \Theta(f(n))$ then $f \in \Theta(g(n))$.

Exercise 2.14 If $g(n) \leq f(n)$ for all $n > 0$ then $\Theta(g(n)) \subseteq O(f(n))$ and $\Theta(f(n)) \subseteq \Omega(g(n))$.

2.1.4 Some Exercises with Solutions

Exercise 2.15 Is 2^{n+1} in $O(2^n)$? Is 2^{2n} in $O(2^n)$?

Proof: Yes. Take $c = 2$ and $n_0 = 1$; then, since $2^{n+1} = 2 \cdot 2^n$ for all n, we have that for each $n \geq 1$, $2^{n+1} \leq c2^n$.

No. There are no c, n_0 such that for each $n \geq n_0$, $2^{2n} \leq c2^n$. To show this, assume there exist such c, n_0. Then $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n$, which implies $2^n \leq c$. But no constant is bigger than 2^n. Contradiction.

Exercise 2.16 Prove that $\log(n!) \in \Theta(n \log n)$.

Proof: First we prove that $\log(n!) \in O(n \log n)$. Since $n! = 1 \cdot 2 \cdot 3 \cdots n - 1 \cdot n \leq n \cdot n \cdots n = n^n$, we have that $\log(n!) \leq \log(n^n) = n \log n$. Then $\log(n!) \in O(n \log n)$.

Now we prove that $\log(n!) \in \Omega(n \log n)$. Since $n! = 1 \cdot 2 \cdot 3 \cdots n - 1 \cdot n \geq \Pi_{i=1}^{n/2} n/2 = (n/2)^{n/2}$, where the $\geq$ step follows after noticing that the last $n/2$ product terms of $n!$ are all greater than $n/2$. Then we have that $\log(n!) \geq \log((n/2)^{n/2}) = (n \log(n/2))/2 = (n \log n - n)/2 \in \Omega(n \log n)$. Then $n \log n = O(\log(n!))$.

Exercise 2.17 Is the function $\lceil \log n \rceil !$ polynomially bounded? Is the function $\lceil \log \log n \rceil !$ polynomially bounded?

Proof: The function $\lceil \log n \rceil !$ is not polynomially bounded. The function $\lceil \log \log n \rceil !$ is polynomially bounded.
We use the fact that proving that a function $f(n)$ is polynomially bounded is equivalent to proving that $\log(f(n)) \in O(\log n)$. To show that this is true, consider that if f is polynomially bounded, then $\exists c, k, n_0$ such that $\forall n \geq n_0$, $f(n) \leq cn^k$, and hence $\log(f(n)) \leq kc \log n$, which implies that $\log(f(n)) \in O(\log n)$. Similarly, it is possible to see that if $\log(f(n)) \in O(\log n)$ then f is polynomially bounded.

We remind the reader that $\log(n!) \in \Theta(n \log n)$ and that $\lceil \log n \rceil \in \Theta(\log n)$. Then we have that $\log(\lceil \log n \rceil !) \in \Theta(\lceil \log n \rceil \cdot \log \lceil \log n \rceil) = \Theta(\log n \cdot \log \log n)$, so $\log n \cdot \log \log n \in \Theta(\log(\lceil \log n \rceil !))$. Suppose that $\lceil \log n \rceil !$ is

polynomially bounded, so $\log(\lceil \log n \rceil !) \in O(\log n)$. Then $\log n \cdot \log\log n \in O(\log n)$, which is of course false. Therefore $\log(\lceil \log n \rceil !) \neq O(\log n)$, and the function $\lceil \log n \rceil !$ is not polynomially bounded. On the other hand, we have that

$$\begin{aligned} \log(\lceil \log\log n \rceil !) &\in \Theta(\lceil \log\log n \rceil \log \lceil \log\log n \rceil) \\ &= \Theta(\log\log n \log\log\log n) \\ &\subseteq O((\log\log n)^2) \\ &\subseteq O(\log n) \end{aligned}$$

Therefore $\log(\lceil \log\log n \rceil !) \in O(\log n)$, so the function $\lceil \log n \rceil !$ is polynomially bounded.

Exercise 2.18 Group the following functions by complexity category: $n \ln n$, $(\lg n)^2$, $5n^2 + 7n$, $n^{5/2}$, $n!$, 4^n, n^n, $n^n + \ln n$, $5^{\lg n}$, $\lg(n!)$, $(\lg n)!$, $\sqrt{n}$, e^n, $8n + 12$, $10^n + n^{20}$.

Proof: Much of the ranking is based on the following facts:

1. Exponential functions grow faster than polynomial functions $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, polynomial functions grow faster than logarithmic functions $\lim_{n\to\infty} \frac{(\lg n)^b}{n^c} = 0$.
2. The base of a logarithm doesn't matter asymptotically, but the base of an exponential and the degree of a polynomial do matter.
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ then $g(n) \in \Theta(f(n))$ if $c > 0$.
   If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then $g(n) \in o(f(n))$.
   If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in o(g(n))$.
4. L'Hopitals rule: If $f(n)$ and $g(n)$ are both differentiable with derivatives $f'(n)$ and $g'(n)$, respectively, and if
   $$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$$
   then
   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$
5. If $\lim_{n\to\infty} lg\left[\frac{f(n)}{g(n)}\right] = \infty$ then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
6. Some useful identities:
   (a) $(\frac{n}{2})^{n/2} \leq n! \leq n^n$
   (b) Stirlings: $n! \sim (n/e)^n \sqrt{2\pi n}$
   (c) $\log_b a = \frac{\log_c a}{\log_c b}$
   (d) $a^{\log_b n} = n^{\log_b a}$

Then the ordering is the following:

n^n and $(n^n + \ln n)$
$n!$
$10^n + n^{20}$
4^n
e^n
$(lgn)!$

$n^{5/2}$
$5^{\lg n}$
$5n^2 + 7n$
$n \log n$ and $\log(n!)$
$8n + 12$
$n^{1/2}$
$\log^2 n$

By fact 3, $\lim_{n\to\infty} \frac{n^n + \ln n}{n^n} = \lim_{n\to\infty} (1 + \frac{\ln n}{n^n}) = 1$

By facts 3 and 6b, $\lim_{n\to\infty} \frac{n^n}{n!} = \lim_{n\to\infty} \frac{n^n}{(n/e)^n\sqrt{2\pi n}} = \lim_{n\to\infty} \frac{e^n}{\sqrt{2\pi n}} = \infty$

By facts 3 and 6a and by noticing that $10^n + n^{20} \leq 2(10^n)$ as n goes to infinity, $\lim_{n\to\infty} \frac{10^n + n^{20}}{n!} < \lim_{n\to\infty} \frac{2(10^n)}{(n/2)^{(n/2)}} = 0$

By fact 3, $\lim_{n\to\infty} \frac{10^n + n^2 0}{4^n} = \lim_{n\to\infty} (\frac{10^n}{4} + \frac{n^2 0}{4^n}) = \infty$

By fact 3, $\lim_{n\to\infty} \frac{4^n}{e^n} = \lim_{n\to\infty} \frac{4}{e}^n = \infty$

By facts 3, 5, and 6a, $\lim_{n\to\infty} \frac{e^n}{(\lg n)!} \leq \lim_{n\to\infty} \frac{e^n}{(\lg n)^{\lg n}} = \lim_{n\to\infty} n \lg e - \lg n \lg\lg n = \infty$

By facts 6a, 6d, and 3, $\lim_{n\to\infty} \frac{(\lg n)!}{n^{5/2}} > \lim_{n\to\infty} \frac{(\lg n/2)^{(\lg n/2)}}{n^{5/2}} = \lim_{n\to\infty} \frac{n^{(\lg\lg n)/2}}{n^{1/2} n^{5/2}} = \infty$

By facts 3 and 5, $\lim_{n\to\infty} \frac{n^{5/2}}{5^{\lg n}} = \lim_{n\to\infty} (5/2)\lg n - \lg 5 \lg n \approx \lim_{n\to\infty} (.18)\lg n = \infty$

By facts 3 and 6c, $\lim_{n\to\infty} \frac{5n^2 + 7n}{5^{\lg n}} \approx \lim_{n\to\infty} (\frac{5}{n^{0.3}} + \frac{7}{n^{1.3}}) = 0$

By facts 3, $\lim_{n\to\infty} \frac{5n^2 + 7n}{n \ln n} = \lim_{n\to\infty} \frac{5n + 7}{\ln n} = \infty$

By facts 3, 6a, and 6c, $\lim_{n\to\infty} \frac{\lg(n!)}{n \ln n} \leq \lim_{n\to\infty} \frac{n \lg n}{(1/\lg e) n \lg n} = \lg e \lim_{n\to\infty} \frac{\lg(n!)}{n \ln n} \geq \lim_{n\to\infty} \frac{n/2 \lg n/2}{(1/\lg e) n \lg n} = \frac{\lg e}{2}$

By facts 3 and 4, $\lim_{n\to\infty} \frac{n \ln n}{8n + 12} = \lim_{n\to\infty} \frac{n/n + \ln n}{8} = \infty$

By fact 3, $\lim_{n\to\infty} \frac{8n + 12}{n^{1/2}} = \lim_{n\to\infty} (8\sqrt{n} + 12/\sqrt{n}) = \infty$

By fact 1, $\lim_{n\to\infty} \frac{(\lg n)^2}{n^{1/2}} = 0$

Exercise 2.19 PROBLEM 2.3(CLR): Ordering by asymptotic growth rates.

You can ignore this problem

Proof: Much of the ranking is based on the fact that: 1) exponential functions grow faster than polynomial functions, which grow faster than logarithmic functions; 2) the base of a logarithm doesn't matter asymptotically, but the base of an exponential and the degree of a polynomial do matter. Moreover, these identities are useful:

$(\log n)^{\log n} = n^{\log \log n}$;

$4^{\log n} = n^2$;

$2^{\log n} = n$;

$2 = n^{1/\log n}$;

$2^{\sqrt{2 \log n}} = n^{\sqrt{2/\log n}}$;

$(\sqrt{2})^{\log n} = \sqrt{n}$;

$\log(n!) = \Theta(n \log n)$.

Then the ordering is the following:

$2^{2^{n+1}}$;

2^{2^n};

$(n+1)!$;

$n!$;

e^n;

$n2^n$;

2^n;

$(3/2)^n$;

$(\log n)^{\log n} = n^{\log \log n}$;

$(\log n)!$;

n^3;

$n^2 = 4 \log n$;

$n \log n$ and $\log(n!)$;

$n = 2^{\log n}$;

$(\sqrt{2})^{\log n}$;

$2^{\sqrt{2 \log n}}$;

$\log^2 n$;

$\log n$;

$\sqrt{\log n}$;

$\log \log n$;

$2^{\log^* n}$;

$\log^* n$ and $\log^*(\log n)$;

$\log(\log^* n)$;

$n^{1/\log n}$.

2.2 Summation

When we analyzed the insertion sort algorithm, we had to calculate the sum $1 + 2 + \cdot + (n-1)$. In fact, it is often the case that the analysis of algorithms reduces to calculating sums, usually more difficult than the one above.

In this section, we'll present a few solved exercises which reflect some usual techniques for calculation of sums, at least for those needed in this course.

2.2.1 Some Exercises with Solutions

Exercise 2.20 Let S_n^k be the sum $1^k + 2^k + 3^k + \cdots + n^k$. Then

1. Calculate S_n^1 and S_n^2, and
2. Show that $S_n^k = O(n^{k+1})$.

Proof:

1. We recall the formulas $(x+1)^2 = x^2 + 2x + 1$ and $(x+1)^3 = x^3 + 3x^2 + 3x + 1$. Then

$$\begin{aligned} S_n^2 &= 1^2 + 2^2 + 3^2 + \cdots + n^2 \\ &= 1 + (1+1)^2 + (2+1)^2 + \cdots + ((n-1)+1)^2 \\ &= 1 + (1^2 + 2\cdot 1 + 1) + (2^2 + 2\cdot 2 + 1) + \cdots + ((n-1)^2 + 2(n-1) + 1) \\ &= (1^2 + 2^2 + \cdots + (n-1)^2) + 2(1 + 2 + \cdots + (n-1)) + n \\ &= (S_n^2 - n^2) + 2(S_n^1 - n) + n \\ &= S_n^2 + 2S_n^1 - n(n+1). \end{aligned}$$

Simplifying by S_n^2, one gets $S_n^1 = \frac{n(n+1)}{2}$.

Similarly,

$$\begin{aligned} S_n^3 &= 1^3 + 2^3 + 3^3 + \cdots + n^3 \\ &= 1 + (1+1)^3 + (2+1)^3 + \cdots + ((n-1)+1)^3 \\ &= 1 + (1^3 + 3\cdot 1^2 + 3\cdot 1 + 1) + (2^3 + 3\cdot 2^2 + 3\cdot 2 + 1) + \\ &\quad \cdots + ((n-1)^3 + 3(n-1)^2 + 3(n-1) + 1) \\ &= (1^3 + 2^3 + \cdots + (n-1)^3) + 3(1^2 + 2^2 + \cdots + (n-1)^2) + \\ &\quad +3(1 + 2 + \cdots + (n-1)) + n \\ &= (S_n^3 - n^3) + 3(S_n^2 - n^2) + 3S_{n-1}^1 + n \\ &= S_n^2 + 3S_n^2 - n^3 - 3n^2 + n + 3\cdot\tfrac{n(n-1)}{2} \\ &= S_n^2 + 3S_n^2 - \tfrac{n(2(n^2+3n-1)-3(n-1))}{2} \\ &= S_n^2 + 3S_n^2 - \tfrac{n(2n^2+3n+1)}{2} \\ &= S_n^2 + 3S_n^2 - \tfrac{n(n+1)(2n+1)}{2}. \end{aligned}$$

Simplifying by S_n^3, one gets $S_n^2 = \frac{n(n+1)(2n+1)}{6}$.

2. We recall that $(n+1)^{k+1} = n^{k+1} + (k+1)n^k + P_{k-1}(n)$, where P_{k-1} is a polynomial of degree $k-1$. Then

$$\begin{aligned} S_n^{k+1} &= 1^{k+1} + 2^{k+1} + 3^{k+1} + \cdots + n^{k+1} \\ &= 1 + (1+1)^{k+1} + (2+1)^{k+1} + \cdots + ((n-1)+1)^{k+1} \\ &= 1 + (1^{k+1} + (k+1)\cdot 1^k + P_{k-1}(1)) + (2^{k+1} + (k+1)\cdot 2^k + P_{k-1}(2)) + \\ &\quad \cdots + ((n-1)^{k+1} + (k+1)\cdot (n-1)^k + P_{k-1}(n-1)) \\ &= (1^{k+1} + 2^{k+1} + \cdots + (n-1)^{k+1}) + (k+1)(1^k + 2^k + \cdots + (n-1)^k) + \\ &\quad +(1 + P_{k-1}(1) + P_{k-1}(2) + \cdots + P_{k-1}(n-1)) \\ &= (S_n^{k+1} - n^{k+1}) + (k+1)(S_n^k - n^k) + O(n^k) \\ &= S_n^{k+1} + (k+1)S_n^k - n^{k+1} + O(n^k) \end{aligned}$$

Simplifying S_n^k, one gets $S_n^k = \frac{n^{k+1}}{k+1} - O(n^k)$, that is, $S_n^k = O(n^{k+1})$.

Exercise 2.21 Calculate the sums $\sum_{k=0}^{n} k2^k$ and $\sum_{k=0}^{n} k^2 2^k$.

Proof: Recall that $\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$. By differentiation, one obtains:

$$\sum_{k=0}^{n} kx^{k-1} = \frac{(n+1)x^n(x-1) - x^{n+1} + 1}{(x-1)^2} = \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2}.$$

Multiplying by x, one gets:

$$\sum_{k=0}^{n} kx^k = \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2}, \qquad (*)$$

and replacing x by 2, $\sum_{k=0}^{n} k2^k = (n-1)2^{n+1} + 2$.

Differentiating $(*)$ again, one obtains

$$\sum_{k=0}^{n} k^2 x^{k-1} = \frac{(n(n+2)x^{n+1} - (n+1)^2 x^n + 1)(x-1)^2 - 2(x-1)(nx^{n+2} - (n+1)x^{n+1} + x)}{(x-1)^4}.$$

Replacing x by 2, we obtain:

$$\sum_{k=0}^{n} k^2 2^{k-1} = n(n+2)2^{n+1} - (n+1)^2 2^n + 1 - 2(n2^{n+2} - (n+1)2^{n+1} + 2) = (n^2 - 2n + 3)2^n - 3,$$

that is, $\sum_{k=0}^{n} k^2 2^k = (n^2 - 2n + 3)2^{n+1} - 6$.

Other Exercises

Exercise 2.22 Calculate $\sum_{k=1}^{n} \frac{1}{2^k}$.

Exercise 2.23 Evaluate $\sum_{k=0}^{\log n} k2^k$, where n is an exact power of 2.

Exercise 2.24 Calculate $\sum_{k=1}^{n} k(k+1)$.

2.3 Recurrence

The running time analysis of most divide and conquer algorithms reduces to solving a recurrence formula. More precisely, if $\mathcal{A}$ is a divide and conquer algorithm which divides the original problem of size n into a subproblems of smaller sizes n_1, n_2, ..., n_a and then combines the solutions of the subproblems into a solution for the original problem in time $f(n)$, then the running time of $\mathcal{A}$ on an input of size n is given by the recurrent formula:

$$T(n) = T(n_1) + T(n_2) + \cdots + T(n_a) + f(n).$$

When the size of the subproblems are small enough, say smaller than a constant n_0, their output can be generated in a number of steps which doesn't depend on the size of the original problem.

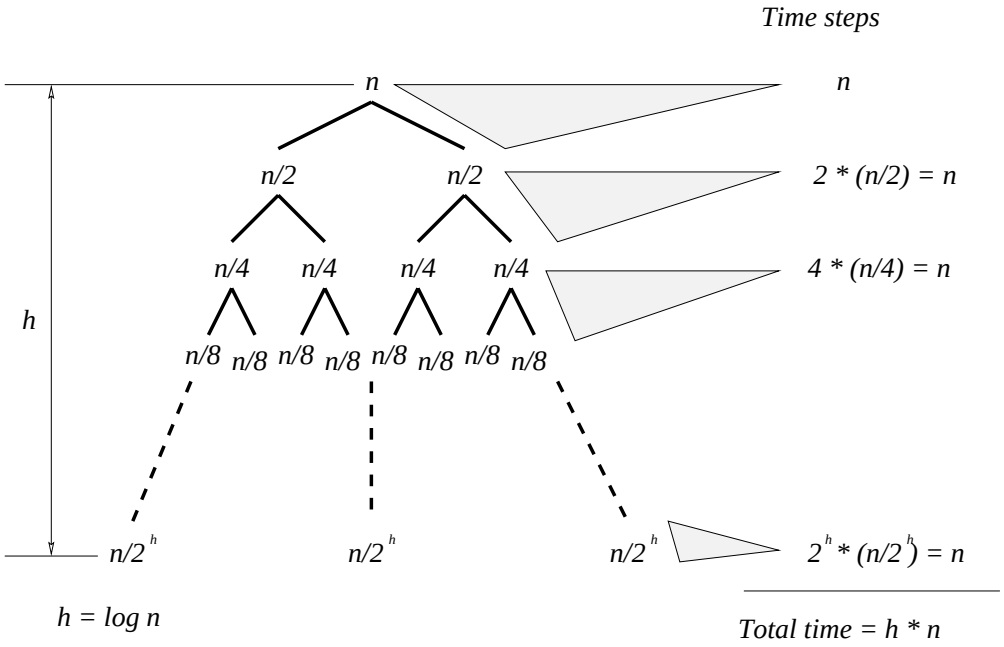
In general, the a subproblems have the same size which is a fraction of n, say n/b, in which case the recurrence formula has the form

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

For example, the analysis of merge sort reduced to solving a recurrence of the same form, for $a = b = 2$ and $f(n) = n$, since the original problem of size n was divided in two subproblems of size $n/2$, whose solutions needed n time steps to combine. We pay more attention to solving this special case of recurrences, even though some of the methods presented below also work for general recurrences. There are three usual methods for solving recurences; we present them in the sequel.

2.3.1 Recursion Tree Method

This is perhaps the most intuitive method for solving recurrences. It basically consists of drawing a tree whose nodes represent the sizes of corresponding subproblems and whose down-edges from a node represent the subproblems of the corresponding problem. Thus each subproblem solved (or "conquered") during the execution of the algorithm belongs to exactly one level in the tree, and each level i contains all those subproblems of "depth" i. There are two important issues which one should consider: the height of the tree and the number of time steps on each level. The solution of the recurrence, or the running time of the corresponding algorithm, is the sum of all time steps at all the levels in the tree. In particular, when the same number of time steps at each level is the same, say $l(n)$, then the solution is $T(n) = h \cdot l(n)$, where h is the height of the tree. The following is a recursion tree for the recurrence $T(n) = 2T(n/2) + n$ of merge sort:



The height of this tree is $h = \log n$ because the size of the subproblems at level h is $n/2^h$, which does not depend on n anymore, so they can be solved in time $\Theta(1)$.

2.3.2 Iterative Method

The iterative method is an "algebrization" of the more intuitive recursion tree method. It consists of unfolding the recurrence for as many steps as needed to "capture" a summation which once solved yields a solution to the original recurrence. The iterative method is more rigurous than the recursion tree method, so the likelyhood to do mistakes is smaller. On the other hand, it may require deeper mathematical knowledge.

Let's use the iterative method to calculate (again) the running time of merge sort:

$$\begin{array}{rcl} T(n) & = & 2T(n/2)+n \\ & = & 2(2T(n/4)+n/2)+n \quad = \quad 2^2T(n/2^2)+2n \quad = \\ & = & 2^2(2T(n/2^3)+n/2^2)+2n \quad = \quad 2^4T(n/2^3)+3n \quad = \\ & \vdots & \\ & = & 2^hT(n/2^h)+hn \qquad \text{(after } h \text{ unfoldings).} \end{array}$$

We keep unfolding the recurrence until $n/2^h$ becomes small enough to not depend on n, for example 1. Then $h = \log n$ (don't worry if h is not an integer), so $T(n) = 2^{\log n}T(1) + n\log n = nT(1) + n\log n \in \Theta(n\log n)$.

2.3.3 Master Method

There is a "magic" theorem solving almost any recurrence:
Master Theorem. Let $T(n) = aT(n/b) + f(n)$ be a recurrence. Then

1. If $f(n) \in O(n^{\log_b a-\epsilon})$ for some $\epsilon > 0$ then $T(n) \in \Theta(n^{\log_b a})$;
2. If $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a}\log n)$;
3. If $f(n) \in \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c > 0$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

In the case of merge sort where $a = b = 2$ and $f(n) = n$, we have that $\log_b a = 1$, so we are in the second case of the theorem. Hence $T(n) \in \Theta(n)$.

The proof of the master theorem is complex and we are not going to approach it in this course. This theorem is very elegant and often easy to use, but we warn the reader that it can sometimes be very tricky, especially its third case. We recommmend you avoid using it whenever possible in this course; use recurrsion trees or iterative methods instead, or use only its second case which is the easiest to check.

2.4 Some Exercises with Solutions

Exercise 2.25 Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + n$ is $\Omega(n\log n)$ by appealing to a recursion tree.

Proof: The shortest path from the root to a leaf in the recursion tree is $n \to (1/3)n \to (1/3)^2n \to \cdots \to 1$. Since $(1/3)^k n = 1$ when $k = \log_3 n$, the height of the part of the tree in which every node has two children is $\log_3 n$. Since the values at each of theses levels of the tree add up to n, the solution to the recurrence is at least $n\log_3 n = \Omega(n\log n)$.

Exercise 2.26 Solve the recurrence $T(n) = T(\sqrt{n}) + 1$.

Proof: First, make the substitution $n = 2^m$ and let $S(m)$ denote $T(2^m)$. Replacing n with 2^m, we obtain the recurrence $T(2^m) = T(2^{m/2}) + 1$, that is, $S(m) = S(m/2) + 1$. Solving this equation using trees, we get the solution $S(m) = O(\log m)$. Replacing $S(m)$ with $T(2^m)$, one obtains $T(2^m) = O(\log m)$, which means that $T(n) = O(\log\log n)$.

Exercise 2.27 A divide an conquer algorithm $\mathcal{A}$ splits the original problem in two related subproblems of size $\sqrt{n}$ and then needs $\log n$ time to combine the solutions of the subproblems into a solution for the original problem. What's the running time of $\mathcal{A}$?

Proof: The recurence for the running time of $\mathcal{A}$ is (1 point)

$$T(n) = 2T(\sqrt{n}) + \log n.$$

Let's solve this recurence (1 point) now. Substituting $n = 2^m$, we get that $T(2^m) = 2T(2^{\frac{m}{2}}) + m$; notice that $m = \log n$. Further, substituting $S(m) = T(2^m)$, we obtain the new recurence $S(m) = 2S(m/2) + m$ which is the same as the one for merge sort, so its solution is $S(m) = m\log m$. Substituting back $S(m) = T(2^m)$ and $m = \log n$, we obtain that $T(n) = \log n \cdot \log\log n$.

Exercise 2.28 The running time of an algorithm A is described by the recurrence $T(n) = 7T(n/2)+n^2$. A competing algorithm has a running time of $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a such that A' is asymptotically faster than A?

Proof: Using the tree method or the master theorem, we can obtain that $T(n) = O(n^{\log_2 7})$ and that $T'(n) = O(n^{\log_4 a})$. Now it is easy to observe that A' is asymptotically faster than A if and only if $\log_4 a < \log_2 7$. This holds when $a < 49$.

Exercise 2.29 The mode of a set of numbers is the number that occurs most frequently in the set. The set (4,6,2,4,3,1) has a mode of 4.

1. Give an efficient and correct algorithm to compute the mode of a set of n numbers;
2. Suppose we know that there is an (unknown) element that occurs $n/2 + 1$ times in the set. Give a worst-case linear-time algorithm to find the mode. For partial credit, your algorithm may run in expected linear time.

Proof:
1. A simple algorithm for this problem is to sort the n numbers, and then traverse the sorted array keeping track of the number with the highest frequency

Inputs: array of numbers S indexed from 1 to n.
Outputs: the number within the set with the highest frequency.

FINDMODE(S,n)

1. MERGESORT(S,n)
2. current $= S[1]$
3. count $= 0$
4. for $i = 2$ to n
5. if ($S[i] =$ current)
6. count = count $+1$
7. else
8. count $= 0$; current $= S[i]$
9. return current

This algorithm's worst-case running time is $\Theta(n \lg n)$ due to the MERGESORT being performed in step 1. The algorithm works because after the set is sorted identical numbers will be adjacent. Therefore, with a linear scan of the array we can find the number with the highest frequency.

b) (Solution 1) A $\Theta(n)$ expected-time algorithm uses the partition procedure from quicksort. A randomly chosen partition element will be used to split the array into two halves. If the partition element is less than the mode (the element which occurs $n/2+1$ times), then the partition element will end up in a position in the range $(1 \ldots (n/2)-2)$. In other words, if the partition element ends up in a position to the left of the center, the mode will be in the set of numbers to the right of the partition element. If the partition element is greater than the mode the partition element will be in position $n/2 + 2$ or greater, and the mode will be in the set of numbers to the left of the partition element. Therefore, after each partition we can recursively call our algorithm on the set of numbers which includes the mode. When the size of the set is 1 then that element is our answer. The running time in the average-case is $T(n) = T(n/2) + n = \Theta(n)$.

Inputs: array of numbers A in which one number appears $n/2 + 1$ times, starting index low, ending index $high$.
Outputs: the number within the set with the highest frequency.

FINDMODE(A,low,$high$)

1. if ($low = high$)
2. return $A[low]$
3. if ($low < high$)
4. $ploc =$ PARTITION(A,low,$high$)
5. if $ploc > \lfloor n/2 \rfloor + 1$
6. FINDMODE(A,low,$ploc - 1$)
7. else

8. FINDMODE(A,$ploc+1$,$high$)

b) (Solution 2) We can use a more exact divide and conquer scheme to get a worst-case running time of $\Theta(n)$. We will divide the set by doing $\lfloor n/2 \rfloor$ pairwise comparisons and only keeping one representative of the pairs which are equal. When n is even, at least one of the $n/2$ comparisons will yield an equality because there are $n/2+1$ of the same element(the mode) in the set. Anytime a number other than the mode is part of an equality, the mode will also be part of another equality. Therefore the number of equalities will be $2k+1$ where k is an integer ≥ 0 and $k+1$ of the equalities will involve the mode. Therefore, we can call our function recursively on the $2k+1$ representatives from the equalities, and the new set will still maintain the attribute of having at least $\lfloor n/2 \rfloor + 1$ of one element. When n is odd, there will be a leftover element after the $\lfloor n/2 \rfloor$ comparisons are done. If the number of representatives left after doing the comparisons is $2k$, where k is an integer ≥ 0, then the left out element should be included because it is the mode. This is because if there are $2k$ equalities from the $\lfloor n/2 \rfloor$ comparisons then at least k of those equalities involve the mode, but the other k could involve another number so the left out element must be included. On the other hand, if there are $2k+1$ equalties then the left out element should not be included in the new set.

Inputs: set of numbers S in which one number appears $n/2+1$ times, the size of the set n
Outputs: the number within the set with the highest frequency.

FINDMODE(S, n)

1. if $n = 1$ then return $S[1]$
2. $S_{new} =$
3. $n_{new} = 0$
4. for $i = 1$ to $\lfloor n/2 \rfloor$ by 2
5. if ($S[i] = S[i+1]$) then
6. $n_{new} = n_{new} + 1$
7. $S_{new} = S_{new} \bigcup S[i]$
8. if (n mod 2) = 1 // n is odd
9. if (count mod 2) = 1
10. $S_{new} = S_{new} \bigcup S[n]$
11. $n_{new} = n_{new} + 1$
12. return FINDMODE(S_{new},n_{new})

Each time FINDMODE is recursively called with a smaller problem which is at most size $\lceil n/2 \rceil$. At each step in the recursion $\lfloor n/2 \rfloor$ comparisons are performed. Therefore, the total running time of the algorithm is

$$T(n) \leq T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor$$

The solution to this recurrence is $T(n) = O(n)$.

b) (Solution 3)

Ignore this solution, it uses notions that you may not know yet.

If an element occurs more than $n/2$ times, then that element will necessarily be the median element. So, the $Select$ algorithm can be used to find that element in deterministic linear time. However, this algorithm has a high constant and so it may be slow in practice.

Exercise 2.30 Professor Diogenes has supposedly identical VLSI chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports wheter it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B Says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

a) Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.
b) Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.
c) Show that the good chips can be identified with $\Theta(n)$ (proportional to n) pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Proof: *Question a)*: Let g be the number of good chips and $n-g$ be the number of bad chips. Also, assume $n-g \geq g$. From this assumption we have that we can always find a set G of good chips and a set B of bad chips of equal size g. Now, assume that the set B of bad chips conspire to fool the professor in the following way: for any test made by the professor, they declare themselves as 'good' and the chips in G as 'bad'. Notice that the chips in G report correct answers and then exhibit a symmetric behaviour: they declare themselves as 'good' and the chips in B as 'bad'. This implies that whichever is the strategy based on the kind of test considered and used by the professor, the behaviour of the chips in G is indistinguishable from the behaviour of the chips in B. This does not allow the professor to determine which chips are good.

Question b): Assume n is even. Then we can match chip in pairs (c_{2i-1}, c_{2i}), for $i = 1, \ldots, n/2$ and test all these pairs. Then we throw away all pairs of chips that do not say 'good, good' and finally we take a chip from each pair. In this way, the final set contains at most n/2 chips and we are still keeping more good chips than bad ones, since all the pairs that we have discarded contain at least a bad chip.

Now, assume n is odd. Then, we test $\lfloor n/2 \rfloor$ pairs constructed as before; this time, however, we don't test one chip, say, c_n. Our final set will contain one chip for every pair tested that reported 'good, good'. Moreover, according to whether the number of such chips has the form $4k+3$ or $4k+1$, we discard chip c_n or we put it into our final set. Now, why this construction works ?

First of all, as in the case in which n is even, by discarding all pairs of chips that report anything but 'good,good', we discard pairs of chips in which at least one is bad, and thus we still keep more good chips than bad ones.

Now, assume that we have discarded the pairs of chips that report anything but 'good,good', and we are left with $4k+3$ chips, for some integer k. Then, in this case there are at least $2k+2$ good chips and thus there are at least $k+1$ pairs of 'good,good' chips, and at most k pairs of 'bad,bad' chips. Thus chip c_n can also be discarded, and only a chip in each of these $2k+2$ pairs is not discarded.

Finally, assume that we have discarded the pairs of chips that report anything but 'good,good', and we are left with $4k+1$ chips, for some integer k. Then, in this case there are at least $2k+1$ good chips.

Now, assume c_n is bad; then there are at least $2k+2$ good chips, that is, $k+1$ pairs of 'good,good' chips; and at most $k-1$ pairs of bad chips. Thus, taking a single chip from every pair, plus c_n gives us at least $k+1$ good chips and at most $(k-1)+1 = k$ bad chips; that is, a majority of good chips and our final set is at most of size $\lceil k/2 \rceil$.

On the other hand, assume c_n is good. Then we have at least k pairs of 'good,good' chips and taking a chip from each pair plus c_n gives us a majority of good chips and a final set of size at most $\lceil k/2 \rceil$.

Question c): If at least $n/2$ of the chips are good, using the answer to question b), we can compute a single good chip. Now, in order to find all good chips, we can just test the good chip with all the others: for each pair of this type, knowing that at least one of the two chips in the pair is good is enough to decide whether the other chip is good or not. This last stage takes $n-1$ more tests. Then, in order to prove that the goof chips can be found with $\Theta(n)$ pairwise tests, we just need to show that the number of tests to find a single good chip is $\Theta(n)$. To show this, we can write the number of tests $T(n)$ that we do in the answer to question b) as

$$T(n) \leq T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor,$$

and $T(1) = 0$. The solution to this recurrence is $T(n) = O(n)$ and can be computed for instance by arguing and proving by induction that $T(n) \leq 2^{\lceil \log n \rceil}$.

Exercise 2.31 This problem has three parts:

1. Solve the recurrence $T(n) = T(n/2) + n$.

2. You are a young scientist who just got a new job in a large team of 100 people (you the 101-st). A friend of yours who you believe told you that you have more honest colleagues than liars, and that that's all what he can tell you, where a *liar* is a person who can either lie or tell the truth, while an *honest* person is one who always tells the truth. Of course, you'd like to know exactly your honest colleagues and the liars, so that you decide to start an investigation, consisting of a series of questions you are going to ask your colleagues. Since you don't wish to look suspicious, you decide to ask only questions of the form "Is Mary an honest person?" and of course, to ask as few questions as possible. Can you sort out all your honest colleagues? What's the minimum number of questions you'd ask in the worst case? You can assume that your colleagues know each other well enough to say if another person is a liar or not. (Hint: Group people in pairs (X,Y) and ask X the question "Is Y honest?" and Y the question "Is X honest?". Analyze all the four possible answers. Once you find an honest person, you can easily find all the others. Challenge: can you solve this enigma asking less than 280 questions in total?)

3. Generalize the strategy above and show that given n people such that less then half are liars, you can sort them out in honest persons and liars by asking $O(n)$ questions.

Proof: The role of 1 is to suggest you that you should reduce the problem to a related subproblem of size half the size of the original problem. Additionally, the recurrence in 1 is closely related to the recurence obtained in 3.

1. Since $T(n) \geq n$, we immeditely get that $T(n) \in \Omega(n)$. On the other hand, since

$$1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^k} \leq 2,$$

unwinding the recurence we obtain:

$$\begin{aligned}
T(n) &= T(\frac{n}{2}) + n \\
&= T(\frac{n}{2^2}) + n(1 + \frac{1}{2}) \\
&\vdots \\
&= T(\frac{n}{2^k}) + n(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}}) \\
&\vdots \\
&\leq T(1) + n(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{\log n - 1}}) \\
&\leq T(1) + 2n \\
&\in O(n).
\end{aligned}$$

Therefore, $T(n) \in \Theta(n)$.

2. Our first goal is to find a honest person. Group your colleagues in pairs (X, Y) and ask X the question "Is Y honest?" and Y the question "Is X honest?". A first important observation is that at least one of X, Y is a liar whenever you get at least a NO answer; then you can remove both X and Y from the set of potential honests, because the remaining $n - 2$ people still verufy the property that more than half are honest. Therefore, we'll keep only those pairs whose answers are YES, noticing that either both people in each group are honest or both are liars.

   Let $T(n)$ be the number of questions (the running time :) you need to ask a group of n people with more honests than liars, in order to find a honest person. We are looking for a recurence for $T(n)$. There can be distinguished three cases, depending on the number of pairs answering (YES,YES) and the parity of people:

   (a) Even number of people grouped in k pairs. Then we can remove one person from each pair, the remaining k people still having the property that more than half are honest. Thus, we obtain the recurrence

$$T(2k) \leq T(k) + 2k,$$

   because we asked $2k$ questions to reduce the problem by half. The worst case is of course when $n = 2k$;

(b) Odd number of people, grouped in even number of pairs $2k$ and one person singled out. Then we can remove one person from each pair, obtaining the recurrence

$$T(4k+1) \leq T(2k+1) + 4k,$$

since we asked questions only the $4k$ people grouped in pairs;

(c) Odd number of people grouped in odd number of pairs $2k+1$ and one person singled out. Then we can safely remove one person from each group together with the singled out one, obtaining the recurrence

$$T(4k+3) \leq T(2k+1) + 4k + 2.$$

Applying the divide and conquer technique described above in which about half the people are removed each step, we can calculate the number of questions needed in the worst case to detect an honest out of 100:

$$\begin{array}{rcrcr} T(100) & \leq & T(50) & + & 100 \\ T(50) & \leq & T(25) & + & 50 \\ T(25) & \leq & T(13) & + & 24 \\ T(13) & \leq & T(7) & + & 12 \\ T(7) & \leq & T(3) & + & 6 \\ T(3) & \leq & T(1) & + & 2 \\ T(1) & = & & & 0 \\ \hline T(100) & \leq & & & 194 \end{array}$$

Once an honest person is found, say X, then we might ask X 99 questions and thus completely correctly divide the 100 people in honests and liars. Therefore, we apparantly need 293 questions in total.

However, it can be done better! We do not need to ask X 99 questions, but to reuse the information we have from previous questions we asked X. For example, during the division of the problem into subproblems, we asked X 6 questions about other 6 people and we got answers YES, so we already knwo that those 6 people are honest. Moreover, each of the known 6 people were asked questions about other people who were previously removed, so those people are also honest. This procedure can be iterated and reduce the number of questions for X by 63! Thus, the total number of questions needed to find all the honest people is 230. I do not see how to do it better.

3. We can upper bound the recurrence above by another more compact recurrence $T(n) \leq T(3n/4) + n$, whose solution is in $O(n)$.

Exercise 2.32 **Problem 1.2-4(CLR)** Consider the problem of evaluating a polynomial at a point. Given n coefficients $a_0, a_1, \ldots, a_{n-1}$ and a real number x, we wish to compute $\sum_{i=0}^{n-1} a_i x^i$. Describe a straightforward $\Theta(n^2)$-time algorithm for this problem. Describe a $\Theta(n)$-time algorithm that uses the following method (called Horner's rule) for rewriting the polynomial:

$$\sum_{i=0}^{n-1} a_i x^i \;=\; (\cdots(a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0.$$

One *cannot* assume the existence of functions that compute the power x^i, but only of functions that compute sums and multiplications.

Proof: The straightforward $\Theta(n^2)$-time algorithm is made of a for loop to compute the power x^i and another for loop to compute the sum $\sum_{i=0}^{n-1} a_i x^i$.

STRAIGHTFORWARD(a, x)

1. $sum \leftarrow a_0$;
2. for $i = 1, \ldots, n-1$,
3. $\quad y \leftarrow 1$;

4. for $j = 1, \ldots, i$,
5. $\quad y \leftarrow y * x$;
6. $sum \leftarrow sum + a_i * y$;
7. return(sum).

In the above algorithm, line 5 is executed $n(n-1)/2$ times, thus the running time is $\Theta(n^2)$. A $\Theta(n)$-time algorithm can be designed using Horner's rule in the following way: in the above formula, the summation is written using n parenthesis; then, the idea is to keep track of the partial results computed in the parenthesis.

HORNER(a, x)

1. $sum \leftarrow a_{n-1}$;
2. for $i = n-2, \ldots, 0$,
3. $\quad sum \leftarrow sum * x + a_i$;
4. return(sum).

Exercise 2.33 Tree isomorphism Give an efficient algorithm to decide , given two rooted binary trees T and T', whether T and T' are isomorphic as rooted trees.

Proof: Let $root$ be the root of T and $root'$ be the root of T', and let L, R, L', R' be the left and right subtrees of T and T' respectively. If T embeds in T', then the isomorphism f must map $root$ to $root'$, and must map every member of L either entirely to L' or entirely to R' and vice versa. Thus, we can define the isomorphism relationship Iso on sub-trees as follows: $Iso(NIL, NIL) = True$, $Iso(x, NIL) = Iso(NIL, x) = False$ if $x \neq NIL$ and otherwise

$$Iso(r_1, r_2) =$$

$$(Iso(Left(r_1), Left(r_2)) AND Iso(Right(r_1), Right(r_2)))$$

OR

$$(Iso(Left(r_1), Right(r_2)) AND Iso(Right(r_1), Right(r_2)).$$

The above immediately translates into a program, and the time, I claim, is $O(n^2)$. The reason is that for any nodes $x \in T, y \in T'$, the above calls itself on x, y at most once (in fact, precisely when x and y are at the same depths of their trees.) Let P be the path from $Root$ to x and P' be that from $Root'$ to y; in a tree there is exactly one' such path. Then, of the four recursive calls at any level, at most one involves nodes from both P and P'. Inductively, at any recursion depth, at most one call compares a sub-tree containing x to one containing y. Since all sub-trees called at depth d of the recursion are depth d in their respective trees, the claim follows. Therefore the total number of calls is at most $O(n^2)$.

Exercise 2.34 Show that the second smallest of n elements can be found with $n + \lceil \log n \rceil - 2$ comparisons in the worst case.

Proof: The smallest of n numbers can be found with $n-1$ comparisons, by conducting a tournament, as follows: compare all the numbers in pairs, consider the set of elements having the smaller element for each pair and repeat the same on this set. The problem is reduced each time by half, and the number of comparison is $n/2 + n/4 + n/8 + \cdots + 1 = n-1$. Graphically, the tournament can be represented as a binary tree, where the root is the minimum and for each node u, his two children u and v were the two elements, such that from their comparison, u came out smaller. Then the second smaller element is clearly one of the elements that have been compared to the smallest element, coming out bigger from such comparison. Since the height of the tree is at most $\lceil \log n \rceil$, finding the second smallest will take at most $\lceil \log n \rceil - 1$ additional comparisons. Thus the total number of comparisons is $n + \lceil \log n \rceil - 2$.

Exercise* 2.35 i-th largest elements Show that, for any constant i, the i-th largest element of an array can be found with $n + O(\log n)$ comparisons.

Proof:

This is a hard problem. Skip it if you get lost ...

First, note that in any comparison algorithm to find the i'th largest in the array, we must compare the i'th largest element to the $i+1$'st largest. (Otherwise, the algorithm would give the same answer if we increased the array position where the $i+1$'st element is to be between the i'th and $i-1$'st largest.)

We'll use this to prove by induction on i that there is a comparison algorithm and constants c_i, d_i so that the algorithm makes a total of at most $n + c_i \log n$ comparisons and no element is compared to more than $d_i \log n$ other elements.

First, the base case is when $i = 1$. We claim we can find the largest element while making $n-1$ comparisons, without comparing any element to more than $\log n$ elements, so the claim is true for $c_1 = 0, d_1 = 1$. We do this by finding the maximum through a divide-and-conquer approach, recursively finding the maximum of the first and last $n/2$ elements, and returning the maximum. The number of comparisons is easily seen to be $n-1$ by solving the recurrence, or by observing that every element but the largest is the smaller element in exactly one comparison the algorithm makes. No element is compared to more than $\log n$ others, since there are $\log n$ recursion levels, and each element is only in one sub-array at any recursion level.

Assume we have an algorithm that finds the i'th largest element in $n + c_i \log n$ comparisons, making no more than $d_i \log n$ comparisons with any particular element. To find the $i+1$'st largest, we run the above algorithm, and then look at all elements compared to the i'th largest, and found to be smaller. There are at most $d_i \log n$ such elements. As observed above, the $i+1$'st largest element must be in this set, and it must be the largest in this set, since all elements are less than the i'th largest. So we find the maximum using the approach in the base case, and output it. This uses at most $d_i \log n$ additional comparisons, so we can let $c_{i+1} = c_i + d_i$. No one element is compared more than $\log(d_i \log n) < \log n$ additional times, so we can let $d_{i+1} = d_i + 1$.

Thus, by induction, for every i there is a comparison algorithm that finds the i'th largest in $n + O(\log n)$ comparisons.

Exercise* 2.36 **k-way merging** Give tight upper and lower bounds in the comparison model for merging k sorted lists with a total of n elements. Lower bounds should be for *any* algorithm, not just the upper bound algorithm.

Proof:

This is another hard problem. Skip it if you find it too difficult

An $O(n \log k)$ algorithm for the problem is to pair the lists umerge each pair, and recursively merege the $k/2$ resulting lists. The time is given by the recurrence $T(n,k) = O(n) + T(n,k/2)$, since the time to perform the merge for each pair of lists is proportional to the sum of the two sizes, so a constant amount of work is done per element. Then unwinding the recurrence gives $\log k$ levels each with $O(n)$ comparisons, for a total time of $O(n \log k)$.

One way to prove the lower bound for any algorithm is to use our lower bound for sorting of $n \log n - O(n)$ and the upper bound for mergesort of $n \log n$ comparisons to reduce the question of sorting n unordered elements to that of merging k sorted lists of total size n as follows.

Let A be a correct algorithm for merging k lists that runs in time $T(n,k)$. We'll prove $T(n,k) \geq n \log k - O(n)$ as follows. Let $A - Sort$ be the algorithm that takes an array of n elements and divides it into k sub-arrays of at most n/k elements each and at most k leftover elements. It sorts each sub-array using mergesort , which takes at most k times $n/k \log(n/k)$. It then places each left-over element within an array, making less than $k(n/k) = n$ comparisons. It calls A on the resulting set of sorted sub-arrays, resulting in a sorted list.

The total number of comparisons for $A - Sort$ is at most $n \log(n/k) + n + T(n,k)$. Since any algorithm for sorting takes at least $n \log n - O(n)$ comparisons, $T(n,k) + n \log(n/k) + n \geq n \log n - O(n)$, so $T(n,k) \geq n \log n - n \log(n/k) - O(n) = n \log n - n \log n + n \log k - O(n) = n \log k - O(n)$.

An alternative method for proving a lower bound is to use a counting argument similar to that used to prove $\Omega(n/lgn)$ for single list sorting. In order for an algorithm to succesfully sort all k-lists of a total of n elements, it must in particular distinguish all partitionings of $1..n$ into k sorted lists of n/k elements each. There are at least $N_{n,k} = (k!)^{n/k}$ such lists. Which is $N_{n,k} > (k/2)^{(k/2)(n/k)}$, and therefore $\log N_{n,k} > cn \log k$ or $\Omega(n \log k)$.

Other Exercises

Exercise 2.37 Solve $T(n) = T(n/2) + 1$.

Exercise 2.38 Solve $T(n) = T(n/2) + n$.

Exercise 2.39 Solve $T(n) = 2T(\sqrt{n}) + 1$.

Exercise 2.40 Solve $T(n) = 3T(n/2) + 2$.

Exercise 2.41 Solve $T(n) = T(2n/3) + 1$.

Exercise 2.42 Solve $T(n) = 4T(n/2) + n^2$.