CS161: Design and Analysis of Algorithms **Summer 2004**

Problem Set #1 Solutions

General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.
- If you use pseudocode in your assignment, please either actually use psuedocode or include a written explanation of your code. The TA does not want to parse through C or JAVA code with no comments.
- If you fax in your problem set, please make sure you write clearly and darkly. Some problem sets were very difficult to read. Also, make sure that you leave enough margins so that none of your work is cut off.
- The version of the Master Theorem that was given in lecture is slightly different from that in the book. We gave case 2 as

$$f(n) \quad = \quad \Theta(n^{\log_b a} \log^k n) \implies T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \text{ for } k \geq 0$$

On exams, using the Master Theorem is normally quicker than other methods. But, remember that cases 1 and 3 only apply when $f(n)$ is polynomially smaller or larger, which is different from asymptotically smaller or larger.

1. **[16 points] Ordering By Asymptotic Growth Rates**

Throughout this problem, you do not need to give any formal proofs of why one function is Ω, Θ, etc... of another function, but please explain any nontrivial conclusions.

(a) **[10 points]** Do problem 3-3(a) on page 58 of CLRS.

Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Answer: Most of the ranking is fairly straightforward. Several identities are helpful:

$$\begin{aligned}
n^{\lg\lg n} &= (\lg n)^{\lg n} \\
n^2 &= 4^{\lg n} \\
n &= 2^{\lg n} \\
2^{\sqrt{2\lg n}} &= n^{\sqrt{2/\lg n}} \\
1 &= n^{1/\lg n} \\
\lg^*(\lg n) &= \lg^* n - 1 \text{ for } n > 1
\end{aligned}$$

In addition, asymptotic bounds for Stirling's formula are helpful in ranking the expressions with factorials:

$$\begin{aligned}
n! &= \Theta(n^{n+1/2}e^{-n}) \\
\lg(n!) &= \Theta(n\lg n) \\
(\lg n)! &= \Theta((\lg n)^{\lg n+1/2}e^{-\lg n})
\end{aligned}$$

Each term gives a different equivalence class, where the $>$ symbol means ω.

$$\begin{array}{cccccccccccc}
2^{2^{n+1}} & > & 2^{2^n} & > & (n+1)! & > & n! & > & e^n & > \\
n\cdot 2^n & > & 2^n & > & (\frac{3}{2})^n & > & \begin{array}{c} n^{\lg\lg n} \\ (\lg n)^{\lg n} \end{array} & > & (\lg n)! & > \\
n^3 & > & \begin{array}{c} n^2 \\ 4^{\lg n} \end{array} & > & \begin{array}{c} n\lg n \\ \lg(n!) \end{array} & > & \begin{array}{c} n \\ 2^{\lg n} \end{array} & > & (\sqrt{2})^{\lg n} & > \\
2^{\sqrt{2\lg n}} & > & \lg^2 n & > & \ln n & > & \sqrt{\lg n} & > & \ln\ln n & > \\
2^{\lg^* n} & > & \begin{array}{c} \lg^*(\lg n) \\ \lg^* n \end{array} & > & \lg(\lg^* n) & > & \begin{array}{c} 1 \\ n^{1/\lg n} \end{array} & & &
\end{array}$$

(b) [**2 points**] Do problem 3-3(b) on page 58 of CLRS.

Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

Answer: $f(n) = (1+\sin n)\cdot 2^{2^{n+2}}$.

(c) [**2 points**] Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n) = o(g_i(n))$.

Answer: $f(n) = 1/n$.

(d) [**2 points**] Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n) = \omega(g_i(n))$.

Answer: $f(n) = 2^{2^{2^n}}$.

2. **[16 points] Recurrences**

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Make your bounds as tight as possible, and justify your answers. You may assume $T(n)$ is constant for sufficiently small n.

(a) **[2 points]** $T(n) = T(9n/10) + n$.

Answer: $T(n) = \Theta(n)$. We have $a = 1$, $b = 10/9$, $f(n) = n$ so $n^{\log_b a} = n^0 = 1$. Since $f(n) = n = \Omega(n^{0+\epsilon})$, case 3 of the Master Theorem applies if we can show the regularity condition holds. For all n, $af(n/b) = \frac{9n}{10} \leq \frac{9}{10}n = cf(n)$ for $c = \frac{9}{10}$. Therefore, case 3 tells us that $T(n) = \Theta(n)$.

(b) **[2 points]** $T(n) = T(\sqrt{n}) + 1$.

Answer: $T(n) = \Theta(\lg \lg n)$. We solve this problem by change of variables. Let $m = \lg n$, and $S(m) = T(2^m)$. Then

$$\begin{aligned} T(n) &= T(n^{1/2}) + 1 \\ T(2^m) &= T(2^{m/2}) + 1 \\ S(m) &= S(m/2) + 1 \end{aligned}$$

We have $a = 1$, $b = 2$, $f(m) = 1$ so $m^{\log_b a} = m^0 = 1$. Since $f(m) = \Theta(1)$, case 2 of the Master Theorem applies and we have $S(m) = \Theta(\lg m)$. To change back to T(n), we have

$$T(n) = T(2^m) = S(m) = \Theta(\lg m) = \Theta(\lg \lg n).$$

Alternatively, we could solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= T(n^{1/2}) + 1 \\ &= (T(n^{1/4}) + 1) + 1 \\ &\dots \\ &= T(n^{1/2^k}) + k \end{aligned}$$

When $n^{1/2^k} \leq 2$, we have that $k \geq \lg \lg n$. Then, $T(n) = \Theta(1) + \lg \lg n = \Theta(\lg \lg n)$.

(c) **[2 points]** $T(n) = 4T(n/2) + n^2 \lg n$.

Answer: $T(n) = \Theta(n^2 \lg^2 n)$. We have $a = 4$, $b = 2$, $f(n) = n^2 \lg n$ so $n^{\log_b a} = n^2$. Since $f(n) = \Theta(n^2 \lg n)$, case 2 of the Master Theorem applies and $T(n) = \Theta(n^2 \lg^2 n)$.

(d) **[2 points]** $T(n) = 5T(n/5) + n/\lg n$.

Answer: $T(n) = \Theta(n \lg \lg n)$. We have $a = 5$, $b = 5$, $f(n) = n/\lg n$ so $n^{\log_b a} = n$. None of the Master Theorem cases may be applied here, since $f(n)$ is neither polynomially bigger or smaller than n, and is not equal to $\Theta(n \lg^k n)$ for any $k \geq 0$. Therefore, we will solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= 5T(n/5) + n/\lg n \\ &= 5(5T(n/25) + (n/5)/(\lg(n/5))) + n/\lg n \end{aligned}$$

$$\begin{aligned}
&= 25T(n/25) + n/\lg(n/5) + n/\lg(n) \\
&\dots \\
&= 5^i T(n/5^i) + \sum_{j=1}^{i-1} n/\lg(n/5^j).
\end{aligned}$$

When $i = \log_5 n$ the first term reduces to $5^{\log_5 n} T(1)$, so we have

$$\begin{aligned}
T(n) &= n\Theta(1) + \sum_{j=1}^{\lg_5 n-1} (n/(\lg(n/5^{j-1}))) \\
&= \Theta(n) + n \sum_{j=1}^{\lg_5 n-1} (1/(\lg n - (j-1)\lg_2 5)) \\
&= \Theta(n) + n(1/\log_2 5) \sum_{j=1}^{\log_5 n-1} (1/(\log_5 n - (j-1))) \\
&= \Theta(n) + n\log_5 2 \sum_{i=2}^{\log_5 n} (1/i).
\end{aligned}$$

This is the harmonic sum, so we have $T(n) = \Theta(n) + c_2 n \ln(\log_5 n) + \Theta(1) = \Theta(n \lg\lg n)$.

(e) **[2 points]** $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

Answer: $T(n) = \Theta(n)$. We solve this problem by guess-and-check. The total size on each level of the recurrance tree is less than n, so we guess that $f(n) = n$ will dominate. Assume for all $i < n$ that $c_1 n \le T(i) \le c_2 n$. Then,

$$\begin{aligned}
c_1 n/2 + c_1 n/4 + c_1 n/8 + kn \le\ & T(n) \le c_2 n/2 + c_2 n/4 + c_2 n/8 + kn \\
c_1 n(1/2 + 1/4 + 1/8 + k/c_1) \le\ & T(n) \le c_2 n(1/2 + 1/4 + 1/8 + k/c_2) \\
c_1 n(7/8 + k/c_1) \le\ & T(n) \le c_2 n(7/8 + k/c_2)
\end{aligned}$$

If $c_1 \ge 8k$ and $c_2 \le 8k$, then $c_1 n \le T(n) \le c_2 n$. So, $T(n) = \Theta(n)$.

In general, if you have multiple recursive calls, the sum of the arguments to those calls is less than n (in this case $n/2 + n/4 + n/8 < n$), and $f(n)$ is reasonably large, a good guess is $T(n) = \Theta(f(n))$.

(f) **[2 points]** $T(n) = T(n-1) + 1/n$.

Answer: $T(n) = \Theta(\lg n)$. We solve this problem by algebraic substitution.

$$\begin{aligned}
T(n) &= T(n-1) + 1/n \\
&= T(n-2) + 1/(n-1) + 1/n \\
&\dots \\
&= \Theta(1) + \sum_{i=1}^{n} 1/i \\
&= \ln n + \Theta(1)
\end{aligned}$$

(g) **[2 points]** $T(n) = T(n-1) + \lg n$.

Answer: $T(n) = \Theta(n \lg n)$. We solve this problem by algebraic substitution.

$$\begin{aligned}
T(n) &= T(n-1) + \lg n \\
&= T(n-2) + \lg(n-1) + \lg n \\
&\dots \\
&= \Theta(1) + \sum_{i=1}^{n} \lg i \\
&= \Theta(1) + \lg(\prod_{i=1}^{n} i) \\
&= \Theta(1) + \lg(n!) \\
&= \Theta(n \lg n)
\end{aligned}$$

(h) **[2 points]** $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

Answer: $T(n) = \Theta(n \lg \lg n)$. We solve this problem by algebraic substitution.

$$\begin{aligned}
T(n) &= \sqrt{n}T(\sqrt{n}) + n \\
&= n^{1/2}(n^{1/4}T(n^{1/4}) + n^{1/2}) + n \\
&= n^{3/4}T(n^{1/4}) + 2n \\
&= n^{3/4}(n^{1/8}T(n^{1/8}) + n^{1/4}) + 2n \\
&= n^{7/8}T(n^{1/8}) + 3n \\
&\dots \\
&= n^{1-1/2^k}T(n^{1/2^k}) + kn
\end{aligned}$$

When, $n^{1/2^k}$ falls under 2, we have $k > \lg \lg n$. We then have $T(n) = n^{1-1/\lg n}T(2) + n \lg \lg n = \Theta(n \lg \lg n)$.

3. **[10 points] Integer Multiplication**

Let u and v be two n-bit numbers, where for simplicity n is a power of 2. The traditional multiplication algorithm requires $\Theta(n^2)$ operations. A divide-and-conquer based algorithm splits the numbers into two equal parts, computing the product as

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

Here, a and c are the higher order bits, and b and d are the lower order bits of u and v respectively. Multiplications are done recursively, except multiplication by a power of 2, which is a simple bitshift and takes $\Theta(n)$ time for an n-bit number. Addition and subtraction also take $\Theta(n)$ time.

(a) **[4 points]** Write a recurrence for the running time of this algorithm as stated. Solve the recurrence and determine the running time.

Answer: $T(n) = 4T(n/2) + \Theta(n) = \Theta(n^2)$. We divide the problem into 4 subproblems (ac, ad, bc, bd) of size $n/2$ each. The dividing step takes constant time, and the recombining step involves adding and shifting n-bit numbers and therefore takes time $\Theta(n)$. This gives the recurrence relation $T(n) = 4T(n/2) + \Theta(n)$. For this case, we have $a = 4$, $b = 2$, $f(n) = n$ so $n^{\log_b a} = n^2$. $f(n) = O(n^{\log_b a-\epsilon})$ so case 1 of the Master Theorem applies. Therefore, $T(n) = \Theta(n^2)$.

(b) **[6 points]** You now notice that $ad+bc$ can be computed as $(a+b)(c+d)-ac-bd$. Why is this advantageous? Write and solve a recurrence for the running time of the modified algorithm.

Answer: $T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{\log_2 3})$. If we write $ad + bc$ as $(a + b)(c + d) - ac - bd$, then we only need to compute three integer multiplications of size $n/2$, namely ac, bd, and $(a + c)(c + d)$. This is advantageous since we replace one multiplication with additions and subtractions, which are less expensive operations. The divide step will now take time $\Theta(n)$, since we need to calculate $a + c$ and $c + d$, and the recombining step will still take $\Theta(n)$. This leads to the recurrence relation $T(n) = 3T(n/2) + \Theta(n)$. We have $a = 3$, $b = 2$, and $f(n) = n$ so $n^{\log_b a} \approx n^{1.585}$. $f(n) = O(n^{1.585-\epsilon})$ so case 1 of the Master Theorem applies. Therefore, $T(n) = \Theta(n^{\lg 3})$.

4. **[20 points] Stable Sorting**

Recall that a sorting algorithm is **stable** if numbers with the same value appear in the output array in the same order as they do in the input array. For each sorting algorithm below, decide whether it is stable or not stable. If the algorithm is stable, give a formal proof of its stability (using loop invariants if necessary). If the algorithm is unstable, suggest a way to modify the algorithm to make it stable. This modification should not alter the fundamental idea of the algorithm or its running time! Explain (informally) why your modifications make the algorithm stable.

(a) **[6 points]** Insertion Sort: pseudocode given on page 17 of CLRS.

```
INSERTION-SORT($A$)
1   for $j \leftarrow 2$ to $length[A]$
2       do $key \leftarrow A[j]$
3           ▷ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$
4           $i \leftarrow j - 1$
5           while $i > 0$ and $A[i] > key$
6               do $A[i+1] \leftarrow A[i]$
7                   $i \leftarrow i - 1$
8           $A[i+1] \leftarrow key$
```

Answer: Insertion sort is stable. We will prove this using the following loop invariant: At the start of each interation of the for loop of lines 1-8, if $A[a] = A[b], a < b \leq j - 1$ and distinct, then $A[a]$ appeared before $A[b]$ in the initial array.

Initialization: Before the first loop iteration, $j = 2$ so there are no distinct

elements $a < b \leq j - 1 = 1$. So, the condition holds trivially.
Maintenance: Given that the property holds before any iteration, we need to show that it holds at the end of the iteration. During each iteration, we insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$. For all $a < b \leq j - 1$, the property holds at the end of the loop since we simply shift elements in the subarray and don't change their respective order. For all $a < b = j$, the property holds at the end of the loop since we insert j after all elements whose value is equal to $A[j]$. This is because the condition for shifting elements to the right is $A[i] > key$ (line 5), so all the elements whose value equals key are not shifted and $A[j]$ is inserted after them. Thus, since we have shown the property is true for all $a < b \leq j - 1$ and $a < b = j$, we have shown it true for all $a < b \leq j$ and so the property is true at the end of the loop.
Termination: When the for loop ends, $j = n + 1$, and the loop invariant states that for all $A[a] = A[b], a < b \leq n$, implies $A[a]$ appeared before $A[b]$ in the initial array. This is the definition of stability and since it applies to the entire array A, insertion sort is stable.

(b) [**6 points**] Mergesort: pseudocode given on pages 29 and 32 of CLRS.

```
MERGE($A, p, q, r$)
1     $n_1 \leftarrow q - p + 1$
2     $n_2 \leftarrow r - q$
3     create arrays $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$
4     for $i \leftarrow 1$ to $n_1$
5         do $L[i] \leftarrow A[p + i - 1]$
6     for $j \leftarrow 1$ to $n_2$
7         do $R[j] \leftarrow A[q + j]$
8     $L[n_1 + 1] \leftarrow \infty$
9     $R[n_2 + 1] \leftarrow \infty$
10    $i \leftarrow 1$
11    $j \leftarrow 1$
12    for $k \leftarrow p$ to $r$
13        do if $L[i] \leq R[j]$
14            then $A[k] \leftarrow L[i]$
15                $i \leftarrow i + 1$
16            else $A[k] \leftarrow R[j]$
17                $j \leftarrow j + 1$
```

```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      then $q \leftarrow \lfloor (p + r)/2 \rfloor$
3          MERGE-SORT($A, p, q$)
4          MERGE-SORT($A, q + 1, r$)
5          MERGE($A, p, q, r$)
```

Answer: Mergesort is stable. We prove this by induction on the fact that Mergesort is stable.

Base Case: When we call merge-sort with indices p and r such that $p \not< r$ (therefore $p == r$), we return the same array. So, calling mergesort on an array of size one returns the same array, which is stable.
Induction: We assume that calling merge-sort on an array of size less than n returns a stably sorted array. We then show that if we call merge-sort on an array of size n, we also return a stably sorted array. Each call to mergesort contains two calls to mergesort on smaller arrays. In addition, it merges these two subarrays and returns. Since we assume that the calls to mergesort on smaller arrays return stably sorted arrays, we need to show that the merge step on two stably sorted arrays returns a stable array. If $A[i] = A[j], i < j$ in the initial array, we need $f(i) < f(j)$ in the new array, where f is the function which gives the new positions in the sorted array. If i and j are in the same half of the recursive merge-sort call, then by assumption, they are in order when the call returns and they will be in order in the merged array (since we take elements in order from each sorted subarray). If i and j are in different subarrays, then we know that i is in the left subarray and j is in the right subarray since $i < j$. In the merge step, we take the elements from the left subarray while the left subarry element is less than or equal to the right subarray element (line 13). Therefore, we will take element i before taking element j and $f(i) < f(j)$, the claim we are trying to prove.
Therefore, Mergesort is stable.

(c) [**8 points**] Quicksort: pseudocode given on page 146 of CLRS.

```
QUICKSORT($A, p, r$)
1   if $p < r$
2      then $q \leftarrow$ PARTITION($A, p, r$)
3         QUICKSORT($A, p, q-1$)
4         QUICKSORT($A, q+1, r$)
```

```
PARTITION($A, p, r$)
1   $x \leftarrow A[r]$
2   $i \leftarrow p-1$
3   for $j \leftarrow p$ to $r-1$
4      do if $A[j] \leq x$
5         then $i \leftarrow i+1$
6            exchange $A[i] \leftrightarrow A[j]$
7   exchange $A[i+1] \leftrightarrow A[r]$
8   return $i+1$
```

Answer: Quicksort is not stable. To make it stable, we can add a new field to each array element, which indicates the index of that element in the original array. Then, when we sort, we can sort on the condition (line 4) $A[j] < x$ OR ($A[j] == x$ AND $index(A[j]) < index(x)$). At the end of the sort, we are guaranteed to have all the elements in sorted order, and for any $i < j$, such that $A[i]$ equals $A[j]$, we will have $index(A[i]) < index(A[j])$ so the sort will be stable.

5. **[22 points] Computing Fibonacci Numbers**

   Recall the Fibonacci numbers as defined in lecture and on page 56 of CLRS. Throughout this problem assume that the cost of integer addition, subtraction, multiplication, as well as reading from memory or writing to memory is $\Theta(1)$, independent of the actual size of the numbers. We will compare the efficiency of different algorithms for computing F_n, the nth Fibonacci number.

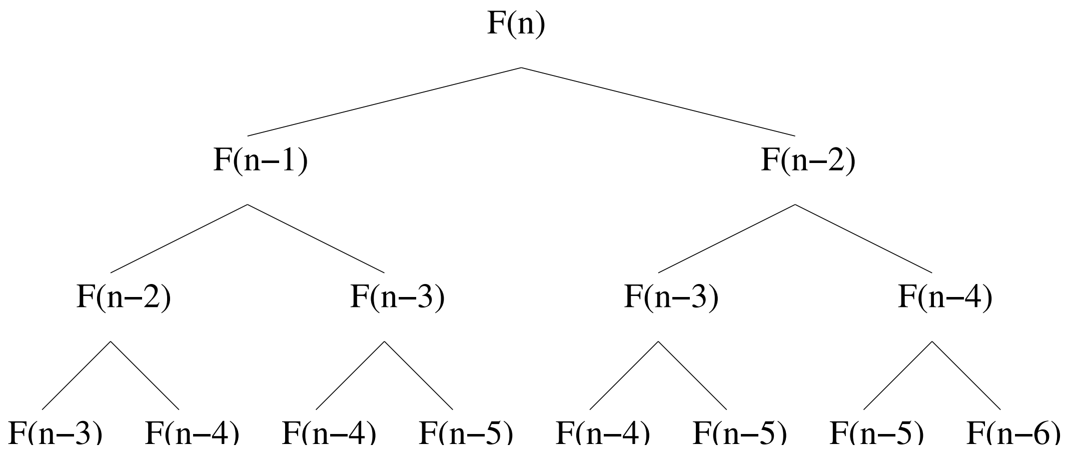
   (a) [**5 points**] First, consider a naive recursive algorithm:

```
int Fib (int n) {
  // recursion
  if (n >= 2) return Fib(n-1) + Fib(n-2);

  // base cases
  if (n == 1) return 1;
  if (n == 0) return 0;

  // error if none of the above returns
}
```

   Sketch a recursion tree for this algorithm. During the computation of Fib(n), how many times is Fib(n-1) called? Fib(n-2)? Fib(n-3)? Fib(2)? Use formula (3.23) on page 56 of CLRS to conclude that this algorithm takes time exponential in n.

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}, \text{ where } \phi = \frac{1+\sqrt{5}}{2} \text{ and } \hat{\phi} = \frac{1-\sqrt{5}}{2} \qquad (1)$$

   Answer:



   We call $Fib(n-1)$ one time during the recursion, $Fib(n-2)$ 2 times, $Fib(n-3)$ 3 times, $Fib(n-4)$ 5 times, ..., $Fib(2)$ F_{n-1} times. Each node in the recurrence tree takes constant time to evaluate, so we need to calculate the number of nodes in the tree. $T(n) = \Theta(\sum_{i=1}^{n} F_i)$. Using formula 3.23 in the book, we get

$$T(n) = \Theta(\sum_{i=1}^{n} \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}})$$

As n grows large, the $\hat{\phi}^i$ term goes to zero, so we are left with

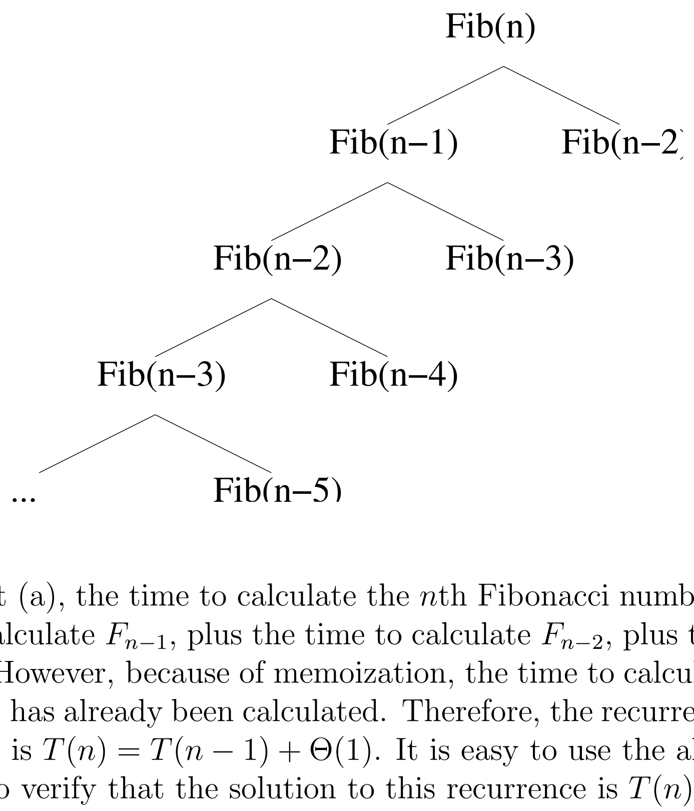
$$T(n) = \Theta(\sum_{i=1}^{n} \frac{\phi^i}{\sqrt{5}}) = \Omega(\phi^n)$$

So, the algorithm takes time exponential in n.

(b) [**5 points**] Notice that our recursion tree grows exponentially because the same computation is done many times over. We now modify the naive algorithm to only compute each Fib(i) once, and store the result in an array. If we ever need Fib(i) again, we simply reuse the value rather than recomputing it. This technique is called **memoization** and we will revisit it when we cover dynamic programming later in the course.

Prune your recursion tree from part (a) to reflect this modification. Considering the amount of computation done at each node is $\Theta(1)$, what is the total amount of computation involved in finding Fib(n)?

Answer:



As in part (a), the time to calculate the nth Fibonacci number F_n is equal to the time to calculate F_{n-1}, plus the time to calculate F_{n-2}, plus the time to add F_{n-1} to F_{n-2}. However, because of memoization, the time to calculate F_{n-2} is constant once F_{n-1} has already been calculated. Therefore, the recurrence for this modified algorithm is $T(n) = T(n-1) + \Theta(1)$. It is easy to use the algebraic substitution method to verify that the solution to this recurrence is $T(n) = \Theta(n)$.

(c) [**4 points**] Prove that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

for $n = 1, 2, 3, \ldots$.

Hint: The last line of the question should immediately suggest what proof technique would be useful here.

Answer: We prove the statement by mathematical induction. For the base case of $n = 1$, we get

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} \tag{2}$$

which certainly holds. Now, we assume the statement holds for $n - 1$. Then,

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} \\ &= \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \end{aligned}$$

Thus, we have shown that the statement holds for all n.

(d) [**8 points**] Use part (c) or formula (3.23) to design a simple divide-and-conquer algorithm that computes the nth Fibonacci number F_n in time $\Theta(\lg n)$.

Answer: Let A be the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ of part (c). To compute the nth Fibonacci number F_n, we can simply exponentiate the matrix A to the nth power and take the element $A^n_{2,1}$. An efficient divide-and-conquer algorithm for this problem uses the notion of **repeated squaring**. In other words, to calculate A^n, observe that

$$A^n = \begin{cases} (A^{n/2})^2 & \text{if } n \text{ is even} \\ (A^{\lfloor n/2 \rfloor})^2 \cdot A & \text{if } n \text{ is odd} \end{cases}$$

This algorithm satisfies the recurrence $T(n) = T(n/2) + \Theta(1)$ because we consider multiplying two 2×2 matrices as a constant time operation and because we have reduced a problem of size n to one of size $\lfloor n/2 \rfloor$. We can solve this recurrence using the master theorem case 2, and arrive at a solution of $T(n) = \Theta(\lg n)$.

6. [**26 points**] **Algorithm Design**

(a) **[10 points]** Design a $\Theta(n \lg n)$-time algorithm that, given an array A of n integers and another integer x, determines whether or not there exist two (not necessarily distinct) elements in A whose sum is exactly x. This is problem 2.3-7 on page 37 of CLRS, slightly reworded for the sake of clarity.

Answer: We first sort the elements in the array using a sorting algorithm such as merge-sort which runs in time $\Theta(n \lg n)$. Then, we can find if two elements exist in A whose sum is x as follows. For each element $A[i]$ in A, set $y = A[i] - x$. Using binary search, find if the element y exists in A. If so, return $A[i]$ and y. If we can't find y for any $A[i]$, then return that no such pair of elements exists. Each binary search takes time $\Theta(\lg n)$, and there are n of them. So, the total time for this procedure is $T(n) = \Theta(n \lg n) + \Theta(n \lg n)$ where the first term comes from sorting and the second term comes from performing binary search for each of the n elements. Therefore, the total running time is $T(n) = \Theta(n \lg n)$.

An alternate procedure to find the two elements in the sorted array is given below:

```
SUM-TO-X($A$)
1   Merge-Sort($A$)
2   $i \leftarrow 1$
3   $j \leftarrow length(A)$
4   while $i \leq j$
5      if $A[i] + A[j]$ equals $x$
6         return $A[i], A[j]$
7      if $A[i] + A[j] < x$
8         then $i \leftarrow i + 1$
9      if $A[i] + A[j] > x$
10        then $j \leftarrow j - 1$
```

We set counters at the two ends of the array. If their sum is x, we return those values. If the sum is less than x, we need a bigger sum so we increment the bottom counter. If the sum is greater than x, we decrement the top counter. The loop does $\Theta(n)$ iterations, since at each iteration we either increment i or decrement j so $j - i$ is always decreasing and we terminate when $j - i < 0$. However, this still runs in time $\Theta(n \lg n)$ since the running time is dominated by sorting.

(b) **[16 points]** The majority element of an array A of length n is the element that appears **strictly more than** $\lfloor n/2 \rfloor$ times in the array. Note that if such an element exists, it must be unique.

Design a $\Theta(n)$-time algorithm that, given an array A of n objects, finds the majority element or reports that no such element exists. Assume that two objects can be compared for equality, read, written and copied in $\Theta(1)$ time, but **no other operations are allowed** (for instance, there is no '<' operator). Explain why your algorithm is correct (no formal proof necessary).

Hint: It is possible to find a candidate solution using only one pass through the array, with the help of an auxiliary data structure such as a stack or a queue. Then with one more pass you can determine whether this candidate is indeed the majority element.

Answer: We will give an algorithm to solve this problem which uses an auxiliary stack and a proof of why it is correct. The following is the pseudocode for the algorithm.

```
FIND-MAJORITY-ELEMENT($A$)
 1   for $i \leftarrow 1$ to $length[A]$
 2       if stack is empty
 3           then push $A[i]$ on the stack
 4       else if $A[i]$ equals top element on the stack
 5           then push $A[i]$ on the stack
 6           else pop the stack
 7   if stack is empty
 8       then return NoMajority
 9   $candidate \leftarrow$ top element on the stack
10   $counter \leftarrow 0$
11   for $i \leftarrow 1$ to $length[A]$
12       if $A[i]$ equals $candidate$
13           then $counter \leftarrow counter + 1$
14   if $counter > \lfloor length[A]/2 \rfloor$
15       then return $candidate$
16   return NoMajority
```

The procedure first generates a candidate object from the array. It finds this element using the stack by looping over the array. If the stack is empty or the current element is the same as the top object on the stack, the element is pushed onto the stack. If the top object of the stack is different from the current element, we pop that object off the stack.

Claim: If a majority element exists, it will be on the stack at the end of this part of the procedure.

Proof: (by contradiction) Call the majority element x and say it appears $i > \lfloor n/2 \rfloor$ times. Each time x is encountered, it is either pushed on the stack or an element (different from x) is popped off the stack. There are $n - i < i$ elements different from x. Assume x is not on the stack at the end of the procedure. Then, each of the i elements of x must have either popped another element off the stack, or been popped off by another element. However, there are only $n - i < i$ other elements, so this is a contradiction. Therefore, x must be on the stack at the end of the procedure.

Notice that this proof does not show that if an element is on the stack at the end of the procedure that it is the majority element. We can construct simple counterexamples ($A = [1, 2, 3]$) where this is not the case. Therefore, after obtaining our candidate majority element solution, we check whether it is indeed the majority element (lines 9-16). We do this by scanning through the array and counting the number of times the element *candidate* appears. We return *candidate* if it appears more than $\lfloor n/2 \rfloor$ times, else we report that no majority element exists.

This procedure scans through the array twice and does a constant amount of computation (pushing and popping elements on the stack take constant time) at each step. Therefore the running time is $T(n) = cn = \Theta(n)$.