CMPUT 204 (B1) – Assignment # 3

DUE DATE: August 8, 2000

Solutions

---

1. *Floors and More Floors*

   Let α and β be real numbers, and let n be a positive integer, show that

   (a) $\lfloor \alpha + n \rfloor = \lfloor \alpha \rfloor + n$,

   (b) $\lfloor \alpha + \beta \rfloor - 1 \le \lfloor \alpha \rfloor + \lfloor \beta \rfloor \le \lfloor \alpha + \beta \rfloor$,

   (c) $\left\lfloor \dfrac{\alpha}{n} \right\rfloor = \left\lfloor \dfrac{\lfloor \alpha \rfloor}{n} \right\rfloor$,

   (d) $\lfloor \alpha \rfloor + \left\lfloor \alpha + \dfrac{1}{n} \right\rfloor + \left\lfloor \alpha + \dfrac{2}{n} \right\rfloor + \cdots + \left\lfloor \alpha + \dfrac{n-1}{n} \right\rfloor = \lfloor n\alpha \rfloor$.

   SOLUTION:

   (a) We can write $\alpha = \lfloor \alpha \rfloor + \theta$, where $0 \le \theta < 1$, so that

   $$\alpha + n = \lfloor \alpha \rfloor + n + \theta$$

   where $\lfloor \alpha \rfloor + n$ is an integer and $0 \le \theta < 1$. Therefore, $\lfloor \alpha + n \rfloor = \lfloor \alpha \rfloor + n$.

   Note: We have used the fact that if $x = k + \theta$ where k is an integer and $0 \le \theta < 1$, then $k = \lfloor x \rfloor$.

   (b) We can write $\alpha = \lfloor \alpha \rfloor + \theta$ where $0 \le \theta < 1$, and $\beta = \lfloor \beta \rfloor + \phi$ where $0 \le \phi < 1$. From part (a), we have

   $$\lfloor \alpha + \beta \rfloor = \lfloor \lfloor \alpha \rfloor + \lfloor \beta \rfloor + \theta + \phi \rfloor = \lfloor \alpha \rfloor + \lfloor \beta \rfloor + \lfloor \theta + \phi \rfloor,$$

   and $0 \le \theta + \phi < 2$, so that

   $$0 \le \lfloor \theta + \phi \rfloor \le 1,$$

   and therefore

   $$\lfloor \alpha \rfloor + \lfloor \beta \rfloor \le \lfloor \alpha + \beta \rfloor \le \lfloor \alpha \rfloor + \lfloor \beta \rfloor + 1,$$

   that is,

   $$\lfloor \alpha + \beta \rfloor - 1 \le \lfloor \alpha \rfloor + \lfloor \beta \rfloor \le \lfloor \alpha + \beta \rfloor.$$

   (c) Again, let $\alpha = \lfloor \alpha \rfloor + \theta$ where $0 \le \theta < 1$, and use the division algorithm to divide $\lfloor \alpha \rfloor$ by n, so that

   $$\lfloor \alpha \rfloor = q \cdot n + r$$

   for some integers q and r with $0 \le r \le n - 1$. This implies that

   $$\frac{\lfloor \alpha \rfloor}{n} = q + \frac{r}{n}$$

   where $0 \le \dfrac{r}{n} \le 1 - \dfrac{1}{n} < 1$, so that $q = \left\lfloor \dfrac{\lfloor \alpha \rfloor}{n} \right\rfloor$.

   On the other hand,

   $$\frac{\alpha}{n} = \frac{\lfloor \alpha \rfloor}{n} + \frac{\theta}{n} = q + \frac{r + \theta}{n}$$

   where $0 \le \dfrac{r + \theta}{n} < 1 - \dfrac{1}{n} + \dfrac{1}{n} = 1$, so that $q = \left\lfloor \dfrac{\alpha}{n} \right\rfloor$.

   Therefore, $\left\lfloor \dfrac{\alpha}{n} \right\rfloor = \left\lfloor \dfrac{\lfloor \alpha \rfloor}{n} \right\rfloor$.

(d) Again, writing $\alpha = \lfloor \alpha \rfloor + \theta$, where $0 \le \theta < 1$, then $0 \le n\theta < n$, so there exists an integer r with

$$r \le n\theta < r+1,$$

where $0 \le r < n$, that is,

$$\frac{r}{n} \le \theta < \frac{r+1}{n}.$$

Now,

$$n\alpha = n\lfloor \alpha \rfloor + n\theta = n\lfloor \alpha \rfloor + r + n\theta - r$$

and $0 \le n\theta - r < 1$, so that $\lfloor n\alpha \rfloor = n\lfloor \alpha \rfloor + r$.

Now, for $0 \le j \le n-1$, we have

$$\alpha + \frac{j}{n} = \lfloor \alpha \rfloor + \theta + \frac{j}{n} \tag{$*$}$$

with $\dfrac{r+j}{n} \le \theta + \dfrac{j}{n} < \dfrac{r+1+j}{n}$.

For those values of j with $0 \le j < n-r$, we have $0 \le \theta + \dfrac{j}{n} < 1$ and from $(*)$, this implies that

$$\left\lfloor \alpha + \frac{j}{n} \right\rfloor = \lfloor \alpha \rfloor \tag{$**$}$$

for the $n-r$ values of j with $0 \le j < n-r$.

Also for those values of j with $n-r \le j \le n-1$, we have $1 \le \theta + \dfrac{j}{n} < 2$, that is, $0 \le \theta + \dfrac{j}{n} - 1 < 1$ and from $(*)$, this implies that

$$\alpha + \frac{j}{n} = \lfloor \alpha \rfloor + 1 + \theta + \frac{j}{n} - 1,$$

so that

$$\left\lfloor \alpha + \frac{j}{n} \right\rfloor = \lfloor \alpha \rfloor + 1 \tag{$***$}$$

for the r values of j with $n-r \le j \le n-1$.

Therefore,

$$\lfloor \alpha \rfloor + \left\lfloor \alpha + \frac{1}{n} \right\rfloor + \left\lfloor \alpha + \frac{2}{n} \right\rfloor + \cdots + \left\lfloor \alpha + \frac{n-1}{n} \right\rfloor = (n-r)\lfloor \alpha \rfloor + r\left(\lfloor \alpha \rfloor + 1\right) = n\lfloor \alpha \rfloor + r = \lfloor n\alpha \rfloor,$$

that is,

$$\sum_{j=0}^{n-1} \left\lfloor \alpha + \frac{j}{n} \right\rfloor = \lfloor n\alpha \rfloor.$$

2. *Sorting Out the Bits*

We assumed so far throughout the course that the keys in a list to be sorted were distinct. However, there are often duplicate keys, and such duplication could make sorting easier, but algorithms that were designed for distinct (or mostly distinct) keys may not take advantage of the duplication. Consider the extreme case, where there are only two possible key values, namely 0 and 1.

(a) What is the order of the number of key comparisons done by Insertion Sort in the worst case? (Describe a worst-case input).

(b) What is the order of the number of key comparisons done by Quicksort in the worst case? (Describe a worst-case input).

(c) Give an efficient algorithm for sorting a list of n keys that may each be either 0 or 1. What is the order of the worst-case running time of your algorithm?

SOLUTION:

(a) As with Insertion Sort operating on a list with n distinct entries, the worst case occurs when the list is already sorted, but in *reverse* order. Also, like the case where all the entries are distinct, the number of comparisons needed here satisfies $W(n) \in O(n^2)$. To find the worst-case input, we count the number of comparisons made when there are k 1's and $n-k$ 0's,

$$\underbrace{1\,1\cdots 1}_{k}\,\underbrace{0\,0\cdots 0}_{n-k}$$

and then find the value of k for which this is largest.

Insertion Sort will make one comparison to see that each of the 1's in positions $2, 3, \ldots, k$ is in the correct position, for a total of $k-1$. The algorithm will then compare the first 0 with all k 1's to insert this 0 into its proper position. For the $n-k-1$ other 0's, it will make $k+1$ comparisons to insert each 0 into its proper position. Thus, the total number of comparisons made to sort the list is

$$f(k) = \underbrace{1+1+\cdots+1}_{k-1} + k + \underbrace{(k+1)+(k+1)+\cdots+(k+1)}_{n-k-1}$$

that is $2k-1+(k+1)(n-k-1) = nk-k^2+n-2$, and this is a maximum when $k = \frac{n}{2}$. In the worst case then, Insertion Sort does at most

$$W(n) = \left\lfloor f\left(\frac{n}{2}\right)\right\rfloor = n-2+\left\lfloor \frac{n^2}{4}\right\rfloor$$

comparisons.

(b) As with Quick Sort operating on a list of n distinct entries, the worst case occurs when the list is already sorted into the correct order (recall that we chose the first element in each sublist as the pivot element). Also, like the case where all the entries are distinct, the number of comparisons needed here satisfies $W(n) \in O(n^2)$. The worst case input is a list consisting of all 0's such as

$$\underbrace{0\,0\cdots 0\,0}_{n}$$

and each time the *Split* procedure is called on a sublist of length k, the list is split into two sublists, one of which is empty, while the other has length $k-1$. The number of comparisons made by this call to *Split* is $k-1$, therefore, the total number of comparisons made by Quick Sort to sort this list is

$$W(n) = n-1+n-2+\cdots+2+1 = \frac{n(n-1)}{2},$$

and again $W(n) \in O(n^2)$.

(c) An efficient algorithm for sorting a list with n elements, all of which are 0 or 1 is as follows.

We keep two pointers (that is, indices) i and j into the list, and initially $i = 1$ and $j = n$. While $i < j$ we continually increment i while $L[i] = 0$, and continually decrement j while $L[j] = 1$. We then interchange $L[i]$ and $L[j]$, and when $i > j$, the list has been sorted.

If i has been incremented k times, then we have compared $L[i]$ to 0 at most $2k$ times and we have compared $L[j]$ to 1 at most $2(n-k)$ times. Therefore, the total number of comparisons made (for all inputs) is at most $2n$, so the order of the worst-case running time is $W(n) \leq 2n \in O(n)$.
An implementation of this algorithm is given below.

```
#include<stdio.h>
main()
{
    int k, N, *B;
    B = (int *)malloc(sizeof *B);
    printf("Enter the number of bits to be sorted:\n");
    scanf("%d",&N);
    printf("Enter %d bits:\n",N);
    for (k = 0; k < N; k++)
        scanf("%d",&B[k]);
    bit_sort(B,N);
    printf("The sorted array is:\n");
    for (k=0; k < N; k++)
        printf("B[%d] = %d\n",k,B[k]);
}
bit_sort(L,n)
int n;
int *L;
{
     int i = 0,j = n - 1,temp;
     while (i < j)
     {
         if (L[i] == 1 && L[j] == 0)
         {
             temp = L[i];
             L[i] = L[j];
             L[j] = temp;
         }
         while (L[i] == 0)
             i++;
         while (L[j] == 1)
             j--;
     }
}
```

3. *The Selection Problem.*

In general, the selection problem involves finding the k^{th}- smallest of a set of n distinct integers. An obvious way to find the k^{th}-smallest integer in $\Theta(n \log n)$ time, is to sort the list using quick sort and return the k^{th} entry. In implementation below, we have modified the `split` routine in the quick sort algorithm to find the k^{th}-smallest entry in a list with n entries.

```
#include <stdio.h>

main()
{
    int i,K,N,*B;
    int KthSmallest;
    printf("Enter the number of integers in the list:\n");
    scanf("%d",&N);
    printf("Enter the value of K: (1 <= K <= %d\n",N);
    scanf("%d",&K);
    B = (int *)malloc(N*sizeof(int));
    printf("Enter %d integers:\n",N);
    for (i = 0; i < N; i++)
    scanf("%d",&B[i]);
    KthSmallest = selection(B,0,N,K-1);
    printf("The %d th smallest entry is: %d\n",K,KthSmallest);
    printf("\n");
    return 0;
} /* main */

int split(int *A, int first, int last)
{   int i, j, pivot, temp;
    pivot = A[first]; j = first;
    for (i = first+1; i <= last; i++)
    {   if (A[i] < pivot)
        {   j++; temp = A[i]; A[i] = A[j]; A[j] = temp;
        }
    }
    temp = A[j]; A[j] = A[first]; A[first] = temp;
    return j;
}/* split */

int selection(int *A, int first, int last, int k)
{
    int position;
    if (first == last) return A[first];
    else
    {   position = split(A,first,last);
        if (k == position) return A[position];
        else if (k < position) return selection(A,first,position-1,k);
        else return selection(A,position+1,last,k);
    }
}/* selection */
```

(a) Explain briefly how the algorithm works.

(b) What is a worst-case input for the algorithm? That is, for what input of list entries and what value of k would the algorithm make the most number of comparisons of list entries?

(c) Let $W(n)$ be the number of comparisons of list entries made in the worst case when *selection* is performed on a list A containing n entries. Find a recurrence relation together with an initial condition satisfied by $W(n)$ for $n \geq 1$. Solve this recurrence relation to find an exact formula for $W(n)$.

(d) Assume that all inputs are equally likely, that is, all values of k are entered with equal frequency and all values of *split* are returned with equal frequency. Let $A(n)$ denote the average number of comparisons of list entries made by *selection* on a list with n entries, find a recurrence relation together with an initial condition satisfied by $A(n)$.

(e) Show that $A(n) \in \Theta(n)$, so that this algorithm does better than quick sort on the average.

SOLUTION:

(a) The function *split* partitions the array so that all keys smaller than the pivot entry come before it in the array and all keys larger than the pivot entry come after it. The function *selection* then continues partitioning the array until the pivot entry is at the k^{th} position. This is done by recursively partitioning the left subarray if k is less than the splitpoint, and by recursively partitioning the right subarray if k is greater than the splitpoint. When k equals the splitpoint, we are done.

(b) As with quick sort, the worst case occurs when the input to the recursive call contains one less entry. This will happen, for example, when the array is already sorted into increasing order and $k = n$.

(c) The selection algorithm has the same worst-case time complexity as quick sort, and if we let $W(n)$ be the number of comparisons of list entries made by the algorithm when executed on a list containing n entries, since no comparisons are done on a list with 1 entry, then $W(n)$ satisfies the discrete initial value problem

$$\begin{aligned} W(n) &= W(n-1) + n - 1, \quad n > 1 \\ W(1) &= 0. \end{aligned}$$

We can solve this in the usual manner,

$$W(n) - W(1) = \sum_{k-2}^{n} [W(k) - W(k-1)] = \sum_{k-2}^{n} (k-1) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2},$$

so that $W(n) = \dfrac{n(n-1)}{2} \in \Theta(n^2)$.

(d) Let p stand for the value of the pivot point, that is, the value returned by the function *split*, this is the index of the pivot entry after the array has been partitioned by *split* and before the first recursive call to *selection* is made.

Let I_i represent the inputs (p and k) to the first recursive call for which the size of the subarray on which this first recursive call is to be made is i, and let $p(I_i)$ be the probability that I_i occurs, for $0 \leq i \leq n-1$. We assume that all such inputs are equally likely to occur.

If $A(n)$ represents the average number of comparisons made by the function *selection* when executed on an array containing n entries, then

$$A(n) = \sum_{i=0}^{n} p(I_i)A(i) + n - 1,$$

since the number of comparisons made by the function *split* is $n-1$. We will calculate the values of $p(I_i)$ by counting the number of inputs which result in a subarray of size i for the first recursive call to *selection*.

There are n inputs for which there is no recursive call, that is for which $i = 0$, namely, $p = k$ for $k = 1, 2, \ldots, n$.

There are 2 inputs for which $i = 1$, that is, the size of the subarray is 1, namely, $p = 2,\ k = 1$, and $p = n - 1,\ k = n$.

There are $2 \cdot 2$ inputs for which $i = 2$, that is, the size of the subarray is 2, namely, $p = 3$, with $k = 1$ or $k = 2$, and $p = n - 2$, with $k = n - 1$ or $k = n$.

There are $2 \cdot 3$ inputs for which $i = 3$, that is the size of the subarray is 3, namely, $p = 3$, with $k = 1, 2, 3$, and $p = n - 3$, with $k = n - 2, n - 1, n$.

Continuing in this manner, there are $2 \cdot i$ inputs for which the size of the subarray is i, namely, $p = i + 1$, with $k = 1, 2, \ldots, i$ and $p = n - i$, with $k = n - i + 1, n - i + 2, \ldots, n$.

$$\vdots$$

There are $2(n - 1)$ inputs for which $i = n - 1$, namely, $p = n$, with $k = 1, 2, \ldots, n - 1$, and $p = 1$, with $k = 2, 3, \ldots, n$.

The total number of inputs is therefore,

$$n + 2(1 + 2 + 3 + \cdots + n - 1) = n + 2\frac{n(n-1)}{2} = n + n(n-1) = n^2,$$

and since all of these inputs are equally likely, we assign the probabilities

$$p(I_i) = \frac{2 \cdot i}{n^2}$$

for $1 \le i \le n - 1$, and $p(I_0) = \dfrac{n}{n^2}$.

Therefore,

$$A(n) = \sum_{i=0}^{n-1} p(I_i)A(i) = \frac{nA(0) + 2\left[A(1) + 2A(2) + \cdots + (n-1)A(n-1)\right]}{n^2} + n - 1, \quad n \ge 1$$

and $A(0) = 0$.

(e) Multiplying the recurrence above by n^2 and replacing n by $n - 1$ we get the following two equations

$$n^2 A(n) = 2\sum_{k=1}^{n-1} kA(k) + n^2(n-1)$$
$$(n-1)^2 A(n-1) = 2\sum_{k=1}^{n-2} kA(k) + (n-1)^2(n-2),$$

subtracting the second equation from the first we have,

$$n^2 A(n) - (n^2 - 1)A(n-1) = (n-1)(3n-2)$$

so that

$$A(n) = \frac{n^2 - 1}{n^2}A(n-1) + \frac{(n-1)(3n-2)}{n^2}$$

for $n \ge 1$. Now, note that for $n \ge 1$, we have

$$\frac{n^2 - 1}{n^2} \le 1 \qquad \text{and} \qquad \frac{(n-1)(3n-2)}{n^2} \le 3,$$

and this implies that

$$A(n) \le A(n-1) + 3$$

for all $n \ge 1$.

Therefore, since $A(0) = 0$, we have

$$A(n) = A(n) - A(0) = \sum_{k=1}^{n} [A(k) - A(k-1)] \le \sum_{k=1}^{n} 3 = 3n,$$

for all $n \ge 1$, that is, $A(n) \in O(n)$.

Also, using the fact that

$$A(n-1) \le 3(n-1)$$

for $n \ge 2$, we have

$$\begin{aligned} A(n) - A(n-1) &= -\frac{1}{n^2}A(n-1) + \frac{(n-1)(3n-2)}{n^2} \\ &\ge -\frac{3(n-1)}{n^2} + \frac{(n-1)(3n-2)}{n^2} \\ &= \frac{(n-1)(3n-5)}{n^2} \\ &= \left(1 - \frac{1}{n}\right)\left[3 - \frac{5}{n}\right] \end{aligned}$$

for $n \ge 2$.

Now,

$$1 - \frac{1}{n} \ge \frac{1}{2} \qquad \text{and} \qquad 3 - \frac{5}{n} \ge \frac{1}{2}$$

for $n \ge 2$, so that

$$A(n) - A(n-1) \ge \frac{1}{4}$$

for $n \ge 2$.

Therefore, since $A(1) = 0$, we have

$$A(n) = A(n) - A(1) = \sum_{k=2}^{n} [A(k) - A(k-1)] \ge \sum_{k=2}^{n} \frac{1}{4} = \frac{n-1}{4} > \frac{n}{8}$$

for $n \ge 2$, and $A(n) \in \Omega(n)$. Thus, $A(n) \in O(n) \cap \Omega(n) = \Theta(n)$.

4. *Divide and Conquer for Powers.*

A divide and conquer algorithm for computing a^n for two integers a and n is given below.

```
int pow(int a, int n)
{
    if (n == 1) return a;
    if (n%2 == 0)
    { int u;
      u = pow(a,n/2);
      return u*u;
    }
    return a*pow(a,n-1);
}
```

(a) Explain briefly how the algorithm works.
(b) Let $T(n)$ denote the number of multiplications (counting squarings as multiplications) performed by a call to *pow(a,n)*. Find a recurrence relation together with an initial condition satisfied by $T(n)$.
(c) Use the recurrence relation in part (b) to show that $T(n) \in \Theta(\log n)$.

SOLUTION:

(a) In order to see how this recursive algorithm works, we look at the binary representation of the positive integer n,

$$n = d_k \cdot 2^k + d_{k-1} \cdot 2^{k-1} + \cdots + d_1 \cdot 2 + d_0,$$

where $d_i = 0$ or 1, for $0 \le i \le k$. Note that this is just the evaluation at $t = 2$ of the polynomial

$$p(t) = d_k \cdot t^k + d_{k-1} \cdot t^{k-1} + \cdots + d_1 \cdot t + d_0,$$

and we can use *Horner's Rule* to evaluate $p(2)$:

$$n = p(2) = (\cdots((d_k \cdot 2 + d_{k-1}) \cdot 2 + d_{k-2}) \cdot 2 + \cdots + d_1) \cdot 2 + d_0.$$

Usually, when we write the binary representation of an integer n, the coefficient d_k of the highest power of 2 is assumed to be 1, for example, we write 1 0 1 0 1 instead of 0 1 0 1 0 1, so that

$$n = p(2) = (\cdots((2 + d_{k-1}) \cdot 2 + d_{k-2}) \cdot 2 + \cdots + d_1) \cdot 2 + d_0.$$

Now it is clear how the recursive method for computing a^n works, we raise a to the power n by using Horner's Rule for the exponent, and working from the inside out. For example,

$$\begin{aligned}
a^{2+d_{k-1}} &= (a^2) \cdot a^{d_{k-1}} \\
a^{(2+d_{k-1})\cdot 2+d_{k-2}} &= \left((a^2) \cdot a^{d_{k-1}}\right)^2 \cdot a^{d_{k-2}} \\
a^{((2+d_{k-1})\cdot 2+d_{k-2})\cdot 2+d_{k-3}} &= \left(\left((a^2) \cdot a^{d_{k-1}}\right)^2 \cdot a^{d_{k-2}}\right)^2 \cdot a^{d_{k-3}} \\
&\ \vdots \\
a^{(\cdots((2+d_{k-1})\cdot 2+d_{k-2})\cdot 2+\cdots+d_1)\cdot 2+d_0} &= \left(\cdots\left(\left(\left((a^2) \cdot a^{d_{k-1}}\right)^2 \cdot a^{d_{k-2}}\right)^2 \cdot a^{d_{k-3}}\right)^2 \cdots a^{d_1}\right)^2 \cdot a^{d_0}
\end{aligned}$$

As an example, note that

$$29 = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1,$$

so that $d_0 = 1$, $d_1 = 0$, $d_2 = 1$, $d_3 = 1$, and $d_4 = 1$, thus,

$$a^{29} = a \cdot a^{28} = a \cdot \left(a^{14}\right)^2 = a \cdot \left(\left(a^7\right)^2\right)^2 = a \cdot \left(\left(a \cdot a^6\right)^2\right)^2 = a \cdot \left(\left(a \cdot \left(a^3\right)^2\right)^2\right)^2 = a \cdot \left(\left(a \cdot \left(a \cdot a^2\right)^2\right)^2\right)^2$$

(b) Note also that the algorithm $pow(a, n)$ comes from the following recursive definition of a^n,

$$a^n = \begin{cases} a & \text{if } n = 1 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\ a \cdot a^{n-1} & \text{otherwise.} \end{cases}$$

Let $T(n)$ denote the number of multiplications (including squarings) performed by the algorithm in a call to $pow(a, n)$.

If $n = 1$, there are no multiplications performed, so that $T(1) = 0$.

If n is even, then $T\left(\frac{n}{2}\right)$ multiplications are performed in the recursive call to $pow\left(a, \frac{n}{2}\right)$, and one multiplication is performed to square $a^{\frac{n}{2}}$.

If $n > 1$ is odd, then $T(n-1)$ multiplications are performed in the recursive call to $pow(a, n-1)$, and one multiplication is performed to obtain $a \cdot a^{n-1}$.

Therefore, $T(n)$ satisfies the following recurrence

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{if } n \text{ is even} \\ T(n-1) + 1 & \text{otherwise.} \end{cases}$$

(c) Now note that if $n > 1$ is odd, then $n - 1$ is even, so that

$$T(n) = T(n-1) + 1 = T\left(\tfrac{n-1}{2}\right) + 2 = T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 2,$$

while if $n > 1$ is even, then

$$T(n) = T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 1,$$

and therefore

$$T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 1 \le T(n) \le T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 2 \qquad (+)$$

for all $n > 1$.

From the left-hand inequality in (+), we have

$$T(n) \ge T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 1 \ge T\left(\left\lfloor \tfrac{n}{2^2} \right\rfloor\right) + 2 \ge T\left(\left\lfloor \tfrac{n}{2^3} \right\rfloor\right) + 3 \ge \cdots \ge T\left(\left\lfloor \tfrac{n}{2^k} \right\rfloor\right) + k.$$

Now, if $2^k \le n < 2^{k+1}$, then $k \le \log_2 n < k + 1$, so that $k = \lfloor \log_2 n \rfloor$, and $\left\lfloor \frac{n}{2^k} \right\rfloor = 1$, so that

$$T(n) \ge T(1) + \lfloor \log_2 n \rfloor = \lfloor \log_2 n \rfloor > \tfrac{1}{2} \log_2 n$$

for $n \ge 2$, and therefore, $T(n) \in \Omega(\log_2 n)$.

From the right-hand inequality in (+), we have

$$T(n) \le T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + 2 \le T\left(\left\lfloor \tfrac{n}{2^2} \right\rfloor\right) + 4 \le T\left(\left\lfloor \tfrac{n}{2^3} \right\rfloor\right) + 6 \le \cdots \le T\left(\left\lfloor \tfrac{n}{2^k} \right\rfloor\right) + 2k,$$

and the same reasoning as above shows that

$$T(n) \le 2 \cdot \lfloor \log_2 n \rfloor \le 2 \cdot \log_2 n$$

for all $n \ge 2$, and therefore, $T(n) \in O(\log_2 n)$. Thus, $T(n) \in \Theta(\log_2 n)$.