Chapter 1: Introduction to Algorithms

Part II

Contents

- Algorithms (Part I)
- Mathematical preliminaries
  - Order of growth (Part I)
  - Summation
  - Recurrence
- Proof of correctness (Part III)
- Analysis of sorting algorithms (Part IV)

Mathematical preliminaries: Summation and recurrence

Summation

Arithmetic series:

A sequence of numbers with a fixed difference between consecutive numbers

Examples:

1, 2, 3, 4, ...

0, 2, 4, 6, ...

Summation

To sum an arithmetic series, multiply the length of the series by the average of the first and the last numbers.

$$1 + 2 + 3 + 4 + 5 = 5 \times (\frac{1+5}{2}) = 15$$

$$2 + 4 + 6 + 8 = 4 \times (\frac{2+8}{2}) = 20$$

$$\sum_{i=1}^{n} i = n(\frac{n+1}{2})$$

Summation

Linearity:

For any real number c and any finite sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$

$$\sum_{k=1}^{n}(c.a_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$$

Summation

We can exploit the linearity property to manipulate summations incorporating asymptotic notation. For example,

$$\sum_{k=1}^{n} \theta(f(k)) = \theta\left(\sum_{k=1}^{n} f(k)\right)$$

Summation

Geometric series:
A series with a constant ratio between successive terms.

Examples

2, 4, 8, 16, ...

$\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ...

Summation

Geometric series:

A series with a constant ratio between successive terms.

Examples

2, 4, 8, 16, ...

½, ¼, ⅛, ...

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

When the summation is infinite and |x| < 1

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}.$$

Analysis of Selection Sort

Basic idea behind Selection sort:

- Given a list of data to be sorted,
  - Select the smallest item and place it in a sorted list
  - Repeat until all of the data are sorted

Selection Sort

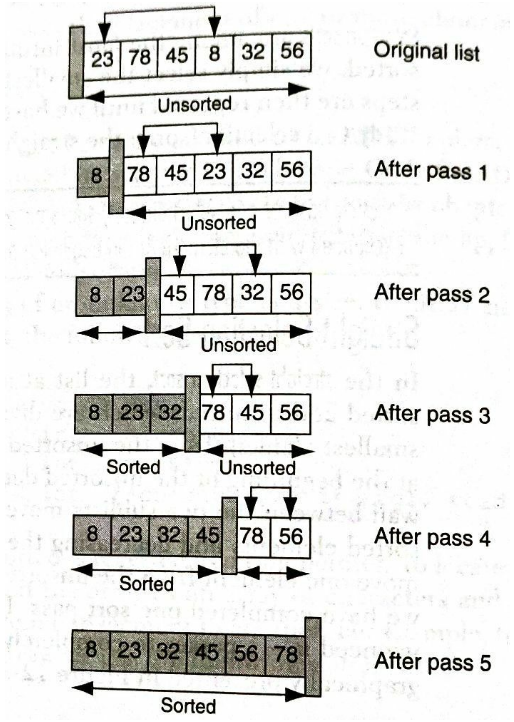
The list at any moment is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall

Steps:

1. Select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data
2. Repeat Step 1 until there is no element in the unsorted sublist

Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed one **sort pass**. Therefore, a list of n elements need n-1 passes to completely rearrange the data

Selection sort



Selection sort

SelectionSort (A) // An unordered array A of size n. Assuming the index of elements starts from 1.

```
for (i = 1 to n-1)
    m = i
    for (j = i + 1 to n)     // Find the minimum element in the unsorted sublist
          if ( A[ j ] < A[m] )
              m = j
          endif
    endfor
    temp = A[i]
    A[i] = A[m]
    A[m] = temp
endfor
```

Analysis of selection sort

Let's assume each execution of the i^{th} line takes time c_i, where c_i is a constant.

	cost	**times**
1. **for** (i = 1 to n-1)	c1	
2. m = i	c2	
3. **for** (j = i + 1 to n)	c3	
4. **if** (A[j] < A[m])	c4	
5. m = j	c5	
6. temp = A[i]	c6	
7. A[i] = A[m]	c7	
8. A[m] = temp	c8	

Analysis of selection sort

		cost	times
1.	**for** (i = 1 to n-1)	c1	$\sum_{i=1}^{n} 1 = n$
2.	m = i	c2	$\sum_{i=1}^{n-1} 1 = n - 1$
3.	**for** (j = i + 1 to n)	c3	$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n+1} 1 = \frac{(n-1)(n+2)}{2}$
4.	**if** (A[j] < A[m])	c4	$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2}$
5.	m = j	c5	Min. 0 times, max. ? times
6.	temp = A[i]	c6	$n - 1$
7.	A[i] = A[m]	c7	$n - 1$
8.	A[m] = temp	c8	$n - 1$

Analysis of selection sort

The running time of the algorithm is the sum of running times for each statement executed.

$$
\begin{aligned}
T(n) &= c_1.n + (c_2 + c_6 + c_7 + c_8)(n-1) + c_3\frac{(n-1)(n+2)}{2} + (c_4 + c_5)\frac{n(n-1)}{2} \\
&= \left(\frac{c_3 + c_4 + c_5}{2}\right)n^2 + \left(c_1 + c_2 + c_3 - (\frac{c_4 + c_5}{2}) + c_6 + c_7 + c_8\right)n - (c_2 + c_3 + c_6 + c_7 + c_8) \\
&= a.n^2 + b.n + c \text{ for constants } a, b, \text{ and } c
\end{aligned}
$$

Analysis of selection sort

When the list is already sorted in ascending order

$$T(n) = c_1.n + (c_2 + c_6 + c_7 + c_8)(n-1) + c_3\frac{(n-1)(n+2)}{2} + c_4\frac{n(n-1)}{2}$$

$$= \left(\frac{c_3 + c_4}{2}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} + c_6 + c_7 + c_8\right)n - (c_2 + c_3 + c_6 + c_7 + c_8)$$

$$= a.n^2 + b.n + c \text{ for constants } a, b, \text{ and } c$$

Time complexity of selection sort = ?

Merge sort

Merge sort uses **divide-and-conquer** strategy

- The original problem is partitioned into simpler sub-problems, each sub problem is considered independently.
- Subdivision continues until sub problems obtained are simple.

Steps:

1. **Divide**: partition the list into two roughly equal parts, S1 and S2, called the left and the right sublists
2. **Conquer**: recursively sort S1 and S2
3. **Combine**: merge the sorted sublists.

Merge sort

Sort the elements in the subarray A[p..r]

MERGE-SORT(A, p, r)

if $p < r$

  $q = \lfloor (p + r)/2 \rfloor$

  MERGE-SORT(A, p, q)

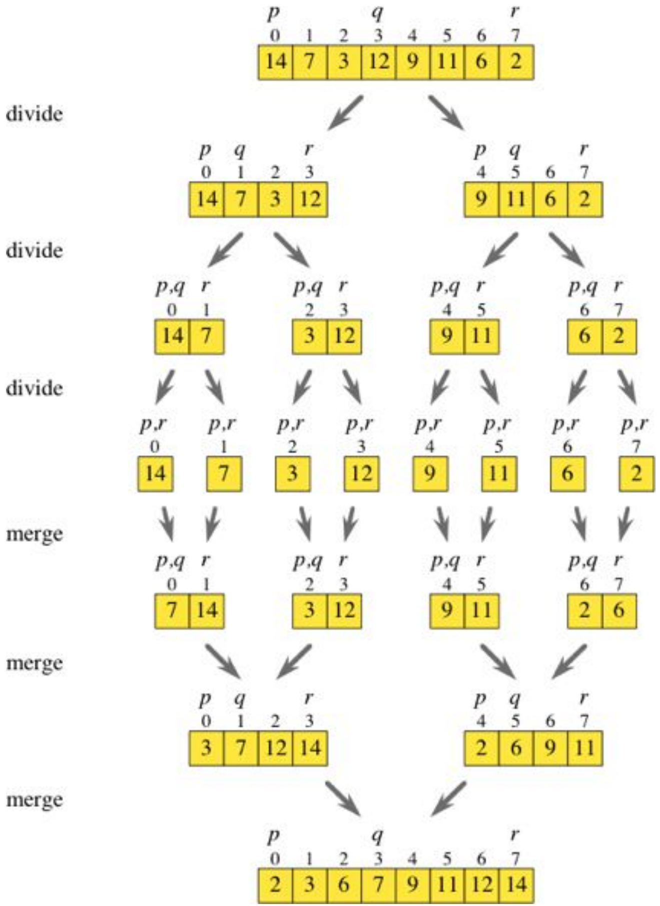
  MERGE-SORT$(A, q + 1, r)$

  MERGE(A, p, q, r)

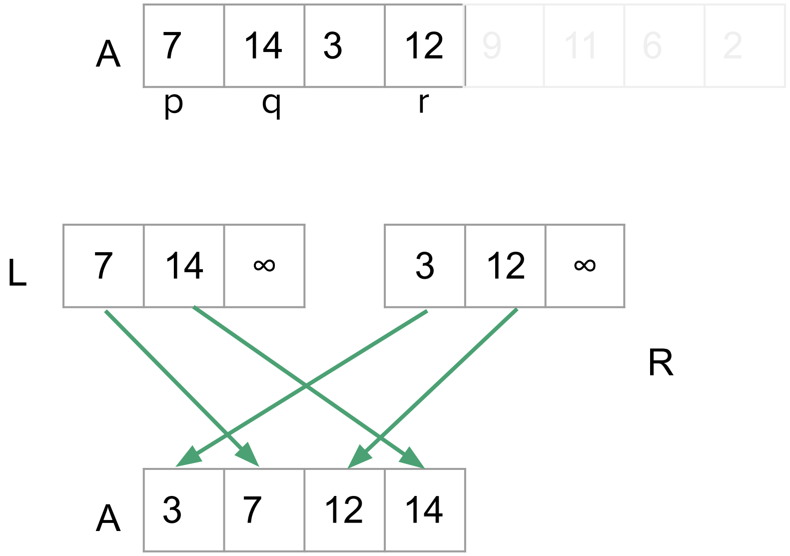
Merge sort

Sort the elements in the subarray A[p..r]

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

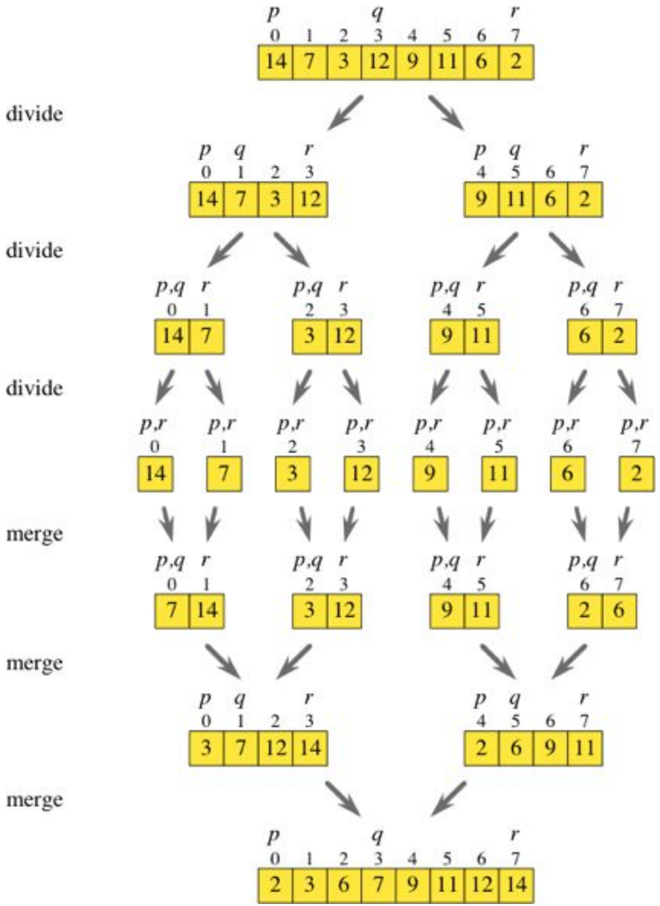


```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i − 1]
for j = 1 to n2
        R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```



Merged sorted list

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i − 1]
for j = 1 to n2
        R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```



Time complexity = ?

Recurrence

When an algorithm contains a recursive call to itself, we can often describe its running time by a **recurrence equation** or **recurrence**.

A recurrence describes the overall running time on a problem of size n in terms of the running time on smaller inputs.

Recurrence

A recurrence for the running time, T(n), (on a problem of size n) of a divide-and-conquer algorithm would be:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \,. \end{cases}$$

where,

- a is the number of subproblems at the division of the problem,
- each subproblem is 1/b the size of the original,
- D(n) is the time to divide the problem into the subproblems, and
- C(n) is the time to combine the solutions to the subproblems into the solution to the original problem

Analysis of merge sort

(For simplification) Let’s assume the original problem size is a power of 2.

- Each divide step yields two subsequences of size exactly $n/2$.
- Merge sort on just one element takes constant time.
- When we have $n>1$ elements, we break down the running time as follows.
  - Divide: Takes constant time. $D(n) = \Theta(1)$
  - Conquer: We recursively solve two subproblems, each of size $n/2$. That is, $a = 2$, $b = 2$.
  - Combine: Merge procedure on an n-element subarray takes time $\Theta(n)$.

Analysis of merge sort

The recurrence for the worst-case running time of merge sort is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

$\Theta(n) + \Theta(1) = \Theta(n + 1) = \Theta(n)$

which can be rewritten as

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps

Solving recurrences

1. Recursion-tree method
2. Substitution method
3. Master method

The recurrence-tree method

This method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.

Example:

The recurrence for the worst-case running time of merge sort is

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$

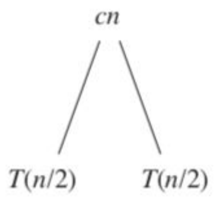
Recurrence tree

$$T(n)$$

We expand this into an equivalent tree representing the recurrence.

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

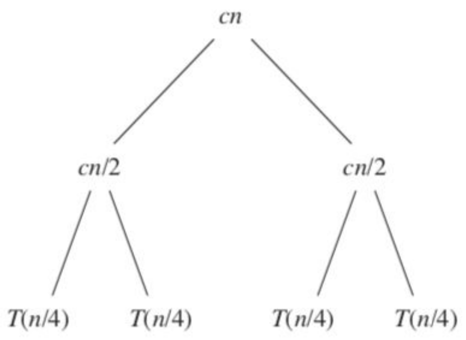
Recurrence tree



We expand this into an equivalent tree representing the recurrence.

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

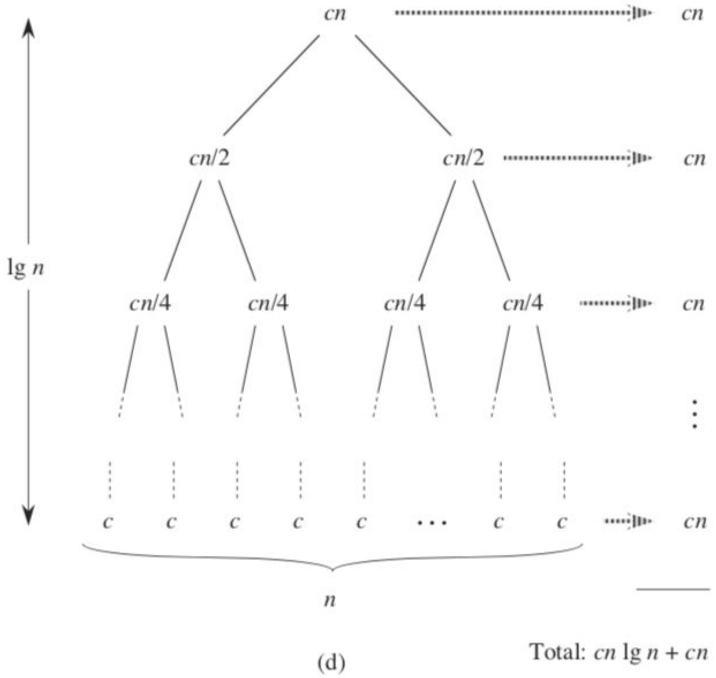
Recurrence tree



We expand this into an equivalent tree representing the recurrence.

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

Recurrence tree



T(n) = Total cost represented by the tree
= number of levels x cn
= (lg n + 1) cn
= cn lg n + cn

T(n) = Θ(n lg n)

The substitution method

In this method, we guess a bound and then use mathematical induction to prove out guess correct.

- Make a guess and then prove the guess inductively (prove by induction).
- If the guess could not be proven correct, make a new guess and prove it correct inductively.

The substitution method

Let's say we have the following recurrence and we want to determine an upper bound on it

$$T(n) = \begin{cases} 7T(\frac{n}{7}) + n & \text{for } n > 1 \\ 1 & \text{for } n = 1 \end{cases}$$

Let's guess that the solution is $T(n) \leq cn$ (i.e., T(n) = O(n)).

We start by assuming that it holds for all positive m < n, in particular m = n/7. That is

$$T(\frac{n}{7}) \leq \frac{cn}{7}$$

The substitution method

We start by assuming that it holds for all positive m < n, in particular m = n/7. That is

$$T(\frac{n}{7}) \leq \frac{cn}{7}$$

This yields

$$T(n) \leq 7(\frac{cn}{7}) + n$$

Or $T(n) \leq (c+1)n$

Unfortunately, we could not prove $T(n) \leq cn$

The substitution method

So, let's make a new guess, $T(n) \leq n\log_7(7n)$ to get the base case of n = 1 right.

We assume this holds true inductively for m < n, in particular m = n / 7

$$T(\frac{n}{7}) \leq \frac{n}{7}\log_7(7\frac{n}{7}) = \frac{n}{7}log_7 n$$

Then we get

$$\begin{aligned} T(n) &\leq 7\left[\tfrac{n}{7}\log_7 n\right] + n \\ &= n\log_7 n + n \\ &= n(\log_7 n + 1) \\ &= n(\log_7 7n) \\ T(n) &\leq n(\log_7 7n) \end{aligned}$$

That verifies our guess.

The master theorem

The master method provides a “cookbook” method for solving recurrences of the form

$$T(n) = a\ T(\frac{n}{b}) + f(n)$$

where a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.

The master theorem

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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

The master theorem (simpler version)

The recurrence $T(n) = a\,T(\frac{n}{b}) + cn^k$ where a, b, c, and k are all constants, solves to

$$
\begin{aligned}
T(n) &\in \theta(n^k) \text{ if } a < b^k \\
T(n) &\in \theta(n^k \lg n) \text{ if } a = b^k \\
T(n) &\in \theta(n^{\log_b a}) \text{ if } a > b^k
\end{aligned}
$$

The master theorem

$$T(n) \in \theta(n^k) \text{ if } a < b^k$$
$$T(n) \in \theta(n^k \lg n) \text{ if } a = b^k$$
$$T(n) \in \theta(n^{\log_b a}) \text{ if } a > b^k$$

Example:

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

Here, $a = 2, b = 2, k = 1$

$$b^k = 2^1 = 2 = a$$

So, it follows from the case a = b^k of the master theorem that

$$T(n) \in \theta(n^k \lg n) = \theta(n \lg n)$$

The master theorem

Exercise:

$$T(n) = 8T(\frac{n}{2}) + 1000n^2$$

$$\boxed{\begin{aligned} T(n) &\in \theta(n^k) \text{ if } a < b^k \\ T(n) &\in \theta(n^k \lg n) \text{ if } a = b^k \\ T(n) &\in \theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}}$$

The master theorem

$$T(n) \in \theta(n^k) \text{ if } a < b^k$$
$$T(n) \in \theta(n^k \lg n) \text{ if } a = b^k$$
$$T(n) \in \theta(n^{\log_b a}) \text{ if } a > b^k$$

Example:

$$T(n) = 8T(\frac{n}{2}) + 1000n^2$$

Here, $a = 8, b = 2, k = 2$

$$b^k = 2^2 = 4 < a$$

So, it follows from the case a > b^k of the master theorem that

$$T(n) \in \theta(n^{\log_b a}) = \theta(n^{\log_2 8}) = \theta(n^3)$$

The master theorem

$$T(n) \in \theta(n^k) \text{ if } a < b^k$$
$$T(n) \in \theta(n^k \lg n) \text{ if } a = b^k$$
$$T(n) \in \theta(n^{\log_b a}) \text{ if } a > b^k$$

Exercise:

$$T(n) = 2T(\frac{n}{2}) + n^2$$