Lecture Note 2

CSCI 6470 Algorithms (Fall 2024)

Liming Cai

School of Computing UGA

August 27, 2024

Chapter 2 Power of Divide-and-Conquer

1. Divide-and-conquer approach
2. More on solving recurrence relations
3. Quick Sort and Algorithm for order statistics
4. Complexity lower bounds

1. Divide-and-conquer approach

- divide-and-conquer is a top-down design approach;
- it breaks the task into several subtasks;
- subtasks are usually solved recursively;

e.g., merge sort, binary search, quick sort, time complexities tend to look like:

$$T(n) = \sum_{i=1}^{k} T(\beta_i n) + B(n)$$

where the task of size n is broken down into k subtasks of the same nature, each with size $\alpha_i n$ for percentage $\beta_i < 1$, $i = 1, 2, \ldots, k$, subject to

$$\sum_{i=1}^{k} \beta_i = 1$$

Some algorithms have $<$ instead of $=$.

1. Divide-and-conquer approach

Merge Sort

```
function MergeSort(L, low, high);
1. if low < high    \\ at least 2 elements
2.     mid = floor((low + high)/2);
3.     MergeSort(L, low, mid);
4.     MergeSort(L, mid+1, high);
5.     MergeTwo(L, low, mid, high);
6.     return;
```

Its time complexity has parameters $\beta_1 = \beta_2 = \frac{1}{2}$, $B(n) = O(n)$. That is

$$T(n) = \begin{cases} a & n \leq 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + bn & n \geq 2 \end{cases}$$

1. Divide-and-conquer approach

Multiplication of two n-bits numbers:

- a recursive strategy:

$$Prod(x, y) = \begin{cases} 2 \times Prod(x, \frac{y}{2}) & \text{if } y \text{ is even} \\ x + 2 \times Prod(x, \lfloor \frac{y}{2} \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

e.g.,

$$9 \times 11 = 9 + 2 \times (9 \times \lfloor \frac{11}{2} \rfloor)$$

Let $T(n)$ be the time complexity for $Prod(x, y)$ when $|y| = n$.

$$T(n) = \begin{cases} a & n = 1 \\ T(n-1) + bn & n \geq 2 \end{cases}$$

$T(n) = O(n^2)$.

1. Divide-and-conquer approach

Multiplication of two n-bits numbers:

- a divide-and-conquer strategy

  (1) split x and y into high and low segments, each with $\frac{n}{2}$ bits;
  (2) $x \times y$ uses 4 $\times$'s on segments plus some additions;

$$\begin{aligned} xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \end{aligned}$$

  e.g., for decimal numbers (based 10 instead)

  Time complexity:

$$T(n) = \begin{cases} a & n = 1 \\ 4T(\frac{n}{2}) + bn & n \geq 2 \longleftarrow \text{why?} \end{cases}$$

  $T(n) = O(n^2)$ can you prove?

1. Divide-and-conquer approach

Multiplication of two *n*-bits numbers:

- a better solution

$$
\begin{aligned}
xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l
\end{aligned}
$$

where with 1 '$\times$' operation

$$(x_h y_h + x_l y_h) = (x_h + x_l)(y_h + y_l) - x_h y_h - x_l y_l$$

$T(n) = 3T(n/2) + O(n)$ leading to

$T(n) = O(n^{1.6})$ can you prove?

2. More on solving recurrence relations

We can prove: for Merge Sort, $T(n) = O(n \log_2 n)$.

Actually, given the recurrence relation for time function

$$T(n) = \sum_{i=1}^{k} T(\beta_i n) + B(n)$$

if $B(n)$ is restricted to $O(n)$, we can prove that $T(n) = O(n \log_2 n)$.

2. More on solving recurrence relations

Theorem. *Assume time function $T(n)$ of some algorithm* has the following recurrence, for fixed number a, b, k, α and β, where $\alpha + \beta = 1$,

$$T(n) = \begin{cases} a & n \leq k \\ T(\alpha n) + T(\beta n) + bn & n > k \end{cases}$$

Then $T(n) = O(n \log_2 n)$, actually, $T(n) = \Theta(n \log_2 n)$.

Proof. Use strong math induction to prove the following claim instead:

There exist $c > 0$ and n_0 such that $T(n) \leq cn \log_2 n$ for all $n \geq n_0$.

2. More on solving recurrence relations

(cont') Proof of the above theorem with strong math induction:

basis: for $n \leq k$, $T(n) = a$. To allow $T(n) = a \leq cn\log_2 n$, it suffices to choose ... what happens here? how to fix it?

assumption: $T(\alpha n) \leq c(\alpha n)\log_2(\alpha n)$ and $T(\beta n) \leq c(\beta n)\log_2(\beta n)$

induction:

$$
\begin{aligned}
T(n) &= T(\alpha n) + T(\beta n) + bn \\
&\leq c(\alpha n)\log_2(\alpha n) + c(\beta n)\log_2(\beta n) + bn \\
&= cn\log_2 n + cn\alpha\log_2\alpha + c\beta n\log_2\beta + bn \\
&= cn\log_2 n - n\big(c\alpha\log_2\frac{1}{\alpha} + c\beta\log_2\frac{1}{\beta} - b\big) \\
&\leq cn\log_2 n \quad \text{when } c \geq \frac{b}{(\alpha\log_2\frac{1}{\alpha} + \beta\log_2\frac{1}{\beta})}
\end{aligned}
$$

3. Quick sort and finding order of statistics

Quick Sort Algorithm

Idea:

- select a pivot element e from the input list L;
- partition list L into two sublists L_h and L_l such that
  $\forall x \in L_h, x > e$
  $\forall x \in L_l, x \leq e$
- recursively sort the two sublists L_h and L_l, separately;

```
function quicksort(L, low, high);
if (low < high)
  k = partition (L, low, high);
  quicksort(L, low, k-1);
  quicksort(L, k+1, high);
return;
```

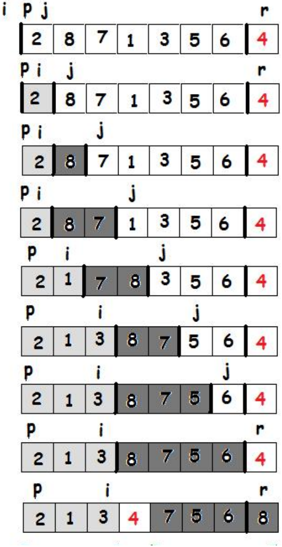
3. Quick sort and finding order of statistics

How does partition work?

```
function partition(L, p, r);

e = L[r];
i = p-1;
for j = p to r-1
  if L[j] <= e
     i = i + 1;
     exchange (L[i], L[j]);
exchange (L[i+1], L[r]);
return i+1
```

Single pass, dynamically 3 regions:
L[p..i]:
L[i+1..j-1]:
L[j..r-1]:
before pivot is in position $\longrightarrow$



3. Quick sort and finding order of statistics

Time complexity, let $n = \texttt{high} - \texttt{low} + 1$;

$$T(n) = \begin{cases} a & n \leq 1 \\ T(|L_h|) + T(|L_l|) + T_P(n) & n \geq 2 \end{cases}$$

where $T_P(n)$ is the time to partition (and to find a pivot from) a list of length n. $T_P(n) = O(n)$

But how big are $|L_h|$ and $|L_l|$?

- note:$|L_h| + |L_l| = n - 1$;
- there is no guarantee that either is a fraction of n in the worst case;
- however, with high probability $|L_h|$ and $|L_l|$ are fractions of n,

  e.g., with 80% chance, $\frac{n}{10} \leq |L_h| \leq \frac{9n}{10}$, why is this true?

3. Quick sort and finding order of statistics

Time complexity, let $n = \texttt{high} - \texttt{low} + 1$;

$$T(n) = \begin{cases} a & n \leq 1 \\ T(|L_h|) + T(|L_l|) + T_P(n) & n \geq 2 \end{cases}$$

where $T_P(n)$ is the time to partition (and to find a pivot from) a list of length n. $T_P(n) = O(n)$

But how big are $|L_h|$ and $|L_l|$?

- note:$|L_h| + |L_l| = n - 1$;
- there is no guarantee that either is a fraction of n in the worst case;
- however, with high probability $|L_h|$ and $|L_l|$ are fractions of n,

  e.g., with 80% chance, $\frac{n}{10} \leq |L_h| \leq \frac{9n}{10}$, why is this true?

  And why is this fact useful?