CMPSCI 311: Introduction to Algorithms

Lecture 22: Randomized and Approximation Algorithms

Akshay Krishnamurthy

University of Massachusetts

Last Compiled: May 2, 2018

Announcements

- HW6 due tomorrow!
- Extra Credit tomorrow as well
- Final on Friday 3:30-5:30 (Marcus Hall 131)
- We are trying our best on grades...
- Please fill out SRTI course evaluations and UCA evaluations.

Remarks on the final

- One problem you have already seen before
  - Either homework or previous exam
- Covers everything fairly equally
  - Big-Oh, Graphs, Greedy, Divide and Conquer, Dynamic Programming, Network Flows, NP-Completeness, Randomized Algs.

Today

- Randomized Median Finding
- Approximate Load Balancing

Randomized Algorithm

- Algorithms that make random choices.
  - Can flip coins, roll dice, etc.
- Two types of randomized algorithms:
  - Fail with some small probability.
  - Always succeed but running time is random.
- How powerful are randomized algorithms?

Median Find

Problem. Given a set of numbers $S = \{a_1, \ldots, a_n\}$ the median is the number in the middle if the numbers were sorted.

- If n is odd then kth smallest element where $k = (n+1)/2$.
- If n is even then kth smallest element where $k = n/2$.

Deterministic algorithm?

- Sort numbers, take kth smallest.
- $O(n \log n)$.

More generally

Problem. Given a set of numbers $S = \{a_1, \ldots, a_n\}$ and number k, return kth smallest number. (Assume no duplicates)

Special cases:

- $k = 1$: minimum element $O(n)$
- $k = n$: maximum element $O(n)$.

Why is it $O(n \log n)$ for $k = n/2$?

Divide and Conquer Algorithm

- Choose splitter (or pivot) $a_i \in S$
- Form sets $S^- = \{a_j : a_j < a_i\}, S^+ = \{a_j : a_j > a_i\}$.

If:

- $|S^-| = k - 1$: a_i is the target.
- $|S^-| \geq k$: recurse on (S^-, k).
- $|S^-| < k - 1$, recurse on $(S^+, k - (|S^-| + 1))$.

Pseudocode

$\textsc{Select}$(S,k):
  Choose splitter $a_i \in S$.
  for each $a_j \in S$ **do**
    Put $a_j \in S^-$ if $a_j < a_i$.
    Put $a_j \in S^+$ if $a_j > a_i$.
  end for
  If $|S^-| = k - 1$, then return a_i.
  If $|S^-| \geq k$, return $\textsc{Select}(S^-, k)$.
  Else, return $\textsc{Select}(S^+, k - (|S^-| + 1))$.

Looks kind of like quicksort. . .

Fact. Algorithm is correct.

How to choose splitter?

We want recursive calls to work on much smaller sets.

- Best case, splitter is the median:

$$T(n) \leq T(n/2) + cn \Rightarrow O(n) \text{ runtime}$$

- Worst case, splitter is largest element:

$$T(n) \leq T(n-1) + cn \Rightarrow O(n^2) \text{ runtime}$$

- Middle case, splitter seperates ϵn elements

$$T(n) \leq T((1-\epsilon)n) + cn$$
$$T(n) \leq cn \left[1 + (1-\epsilon) + (1-\epsilon)^2 + \ldots\right] \leq \frac{cn}{\epsilon}$$

How can we stay close to the best case?

Randomized Splitters

Idea. Choose splitter uniformly at random.

Analysis. Phase j when $n(3/4)^{j+1} \leq |S| \leq n(3/4)^j$.

- **Claim.** Expect to stay in phase j for two rounds.
  - Call splitter *central* if separates $1/4$ fraction of elements.
  - $\Pr[\text{central splitter}] = 1/2$.
  - If X is number of attempts until central splitter,

$$\begin{aligned}\mathbf{E}[X] &= \sum_{j=1}^{\infty} j \Pr[X = j] = \sum_{j=1}^{\infty} jp(1-p)^{j-1} \\ &= \frac{p}{1-p} \sum_{j=1}^{\infty} j(1-p)^j = \frac{p}{1-p} \frac{(1-p)}{p^2} \\ &= \frac{1}{p}\end{aligned}$$

Analysis

- Let Y be a r.v. equal to number of steps of the algorithm
- $Y = Y_0 + Y_1 + Y_2 + \ldots$ where Y_j is steps in phase j
- One iteration in phase j takes $cn(3/4)^j$ steps.
- $\mathbf{E}[Y_j] \leq 2cn(3/4)^j$ since expect two iterations.

$$\begin{aligned}\mathbf{E}[Y] &= \sum_j \mathbf{E}[Y_j] \leq \sum_j 2cn(3/4)^j \\ &= 2cn \sum_j (3/4)^j \leq 8cn\end{aligned}$$

Theorem

Expected running time of $\textsc{Select}$*(n,k) is* $O(n)$.

Applications

- Randomized median find in expected linear time

Quicksort (Sketch)

- Choose pivot at random. Form S^-, S^+.
- Recursively sort both.
- Concatenate together.

Theorem. Quicksort has expected $O(n \log n)$ time.

Approximation Algorithms

- We've seen important problems that are NP-complete. For these problems, should we just give up? No.
- Perhaps we can *approximate* them. For example, for a minimization problem can we design an algorithm such that whenever we run the algorithm we can guarantee that

$$\frac{\text{value of our solution}}{\text{value of optimum solution}} \leq \alpha$$

for some value of $\alpha \geq 1$. Such an algorithm is called an α-approximation algorithm.

Load Balancing

- **Input.** There are m machines and n jobs $\{1, 2, \ldots, n\}$ to be done. The time it takes to do each job is $t_1, t_2, \ldots, t_n$.
- **Goal.** Divide the jobs between the m machines such that no machine does too much work, i.e., if $S_1, \ldots, S_m \subset \{1, 2, \ldots, n\}$ are the set of jobs done by each machine then we want to minimize:

$$T = \max\left(\sum_{i \in S_1} t_i, \ldots, \sum_{i \in S_m} t_i\right)$$

i.e., the time taken by the last machine to finish their jobs.

- We say the total amount of time of jobs allocated to a machine is its load

A Simple Algorithm

- For $i = 1$ to n:
  - Assign job to the machine who currently has the smallest load.

Analysis: Part 1

- Let T^* be smallest possible value $\max\left(\sum_{i \in S_1} t_i, \ldots, \sum_{i \in S_m} t_i\right)$
- Lemma 1: $T^* \geq t_i$ for all $i = 1, 2, \ldots, n$.
- Proof: Some machine needs to do the ith job and that machine is going to take at least t_i time. The max time taken is at least the time this machine spends.
- Lemma 2: $T^* \geq (\sum_{i=1}^n t_i)/m$.
- Proof: If every machine took $< (\sum_{i=1}^n t_i)/m$ time, then the total amount of work done is $< \sum_{i=1}^n t_i$. But this is impossible since all the jobs need to be done.

Analysis: Part 2

- When a machine is assigned job i by the algorithm,

$$\text{its new load} = \text{its old load} + t_i$$

- Recall that we assigned the job to the machine with the smallest current load. The smallest current load is at most $(\sum_{i=1}^n t_i)/m$.
- Hence, by appealing to Lemma 1 and Lemma 2,

$$\text{its new load} < (\sum_{i=1}^{n} t_i)/m + t_i \leq 2T^*$$

i.e., a machine can never be assigned more than a load of $2T^*$.

- Hence, the algorithm is a 2-approximation.

An Improved Algorithm

- Sort the jobs such that $t_1 \geq t_2 \geq t_3 \geq \ldots \geq t_n$
- For $i = 1$ to n:
  - Assign job to the machine who currently has the smallest load.

Analysis: Part 1

- Let T^* be smallest possible value $\max\left(\sum_{i \in S_1} t_i, \ldots, \sum_{i \in S_m} t_i\right)$
- Lemma 3: $T^* \geq 2t_{m+1}$.
- Proof: Some machine must do at least two of the jobs $\{1, 2, \ldots, m+1\}$, say jobs i and j. That machine takes at least $t_i + t_j \geq 2t_{m+1}$ time.

Analysis: Part 2

- When a machine is assigned job i by the algorithm,

$$\text{new load} = \text{ old load} + t_i$$

- Recall that we assigned the job to the machine with the smallest current load. The smallest current load is at most $(\sum_{i=1}^{n} t_i)/m$ and is 0 if $i \leq m$.
- Hence, if $i \leq m$ then by appealing to Lemma 1,

$$\text{new load} = 0 + t_i \leq T^*$$

- Hence, if $i \geq m+1$, by appealing to Lemma 2 and Lemma 3,

$$\text{new load} < (\sum_{i=1}^{n} t_i)/m + t_i \leq T^* + t_{m+1} \leq T^* + T^*/2 = 1.5T^*$$

- Hence, the algorithm is a 1.5-approximation since no machine can ever be assigned more than 1.5 times the optimum.