no lecture on Thursday this week

next and last lecture on Monday next week

Algorithms: Correctness & Efficiency – G52ACE

Randomised Algorithms: Quicksort (guest lecture, covering for Prof Logan)

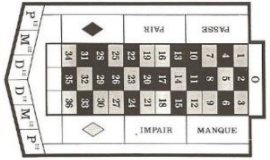
Thomas Gärtner

University of Nottingham
UK | CHINA | MALAYSIA

Professor of Data Science
School of Computer Science, University of Nottingham








Quicksort

<u>**QuickSort**</u>(S)
input: set S of numbers
output: sorted list of elements from S
0: **if** $|S| \leq 1$ **then return** [e **for** e **in** S]
1: pick random element e from S (uniformly)
2: split S into subsets:

- set G of elements greater than e
- set L of elements less than e

3: **return** *QuickSort(L)* ⊕ [e] ⊕ *QuickSort(G)*

Worst case running time is $O(n^2)$.

BUT what should we **expect** to happen?

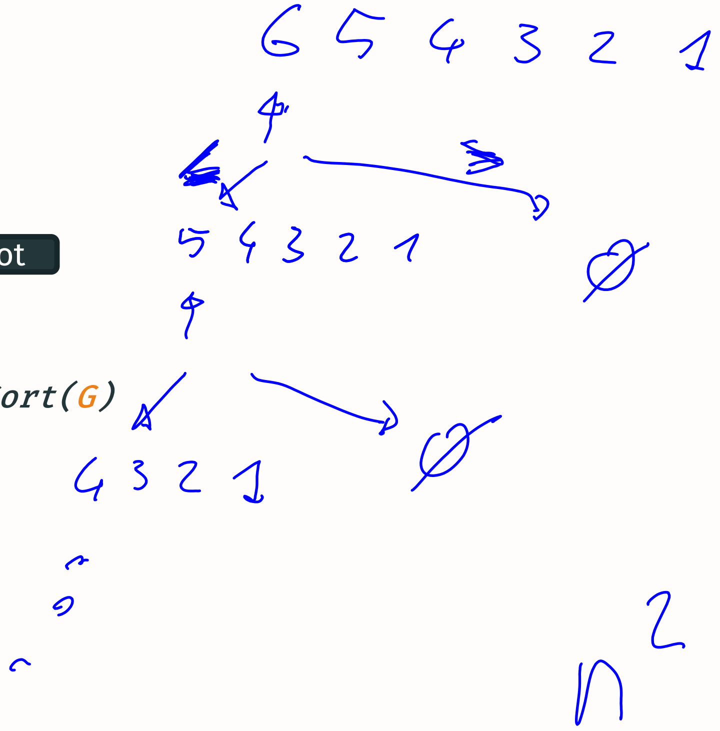
Quicksort

QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: **if** $|S| \leq 1$ **then return** [e **for** e **in** S]
1: pick first element e from S

splitter / pivot

2: split S into subsets:
- set G of elements greater than e
- set L of elements less than e

3: **return** *QuickSort(L)* $\oplus$ [e] $\oplus$ *QuickSort(G)*



Lower bound: How many comparisons will we need?

How many comparisons will any comparison-based sorting algorithm need to make?

There are n! different orderings for n elements
Each set of comparisons partitions the orderings into sets of “compatible” orderings

Example $\{a, b, c\}$, 3!=6

$a < b$

$b < c$

✓	✓	✓	a b c
X	X	✓	a c b
X	✓	X	b a c
X	✓	X	b c a
X	X	✓	c a b
X	X	X	c b a

Lower bound: How many comparisons will we need?

How many comparisons will any comparison-based sorting algorithm need to make?

There are n! different orderings for n elements
Each set of comparisons partitions the orderings into sets of “compatible” orderings

Example $\{a, b, c\}$, 3!=6
$a \prec b$ is compatible with 3 orders
$a \succ b$ is compatible with 3 orders

Lower bound: How many comparisons will we need?

How many comparisons will any comparison-based sorting algorithm need to make?

There are n! different orderings for n elements
Each set of comparisons partitions the orderings into sets of "compatible" orderings

2^k "buckets"

Example $\{a, b, c\}$, 3!=6
$a \prec b$ is compatible with 3 orders: $a \prec b \prec c, a \prec c \prec b, c \prec a \prec b$
$a \succ b$ is compatible with 3 orders: $b \prec a \prec c, b \prec c \prec a, c \prec b \prec a$

$a \prec b, b \prec c$ uniquely determine an order
$a \prec b, a \prec c$ do not uniquely determine an order

a b c
a c b

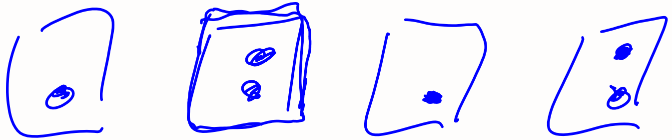
Lower bound: How many comparisons will we need?

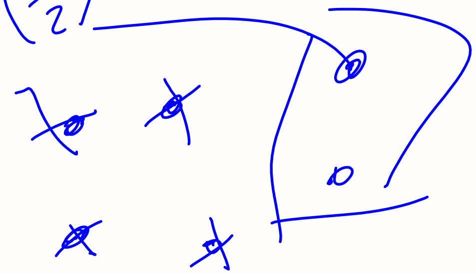
How many comparisons will any comparison-based sorting algorithm need to make?

There are n! different orderings for n elements

Each set of comparisons partitions the orderings into sets of “compatible” orderings

If $2^k < n!$, by the pigeonhole principle, there are orderings that we can't distinguish with k comparisons. -> We need k such that $\boxed{2^k \geq n!} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$





$$\begin{array}{cccccccc} 1 & 1 & 1 & & 1 & \frac{n}{2} & \frac{n}{2} & \frac{n}{2} \\ 1 \cdot 2 & \cdot\, 3 & \cdot\, 4 & \cdots & & \frac{n}{2} \cdots & \cdot\,(n-1) & \cdot\, n \end{array}$$

Lower bound: How many comparisons will we need?

How many comparisons will any comparison-based sorting algorithm need to make?

There are n! different orderings for n elements

Each set of comparisons partitions the orderings into sets of “compatible” orderings

If $2^k < n!$, by the pigeonhole principle, there are orderings that we can’t distinguish with k comparisons. -> We need *k* such that $2^k \geq n!$, for a *rough estimate* we can use $n! \geq \left({}^{n}/_{2}\right)^{{}^{n}/_{2}}$

Therefore $k \geq {}^{n}/_{2} \log_2 {}^{n}/_{2}$

$$2^k \geq \left(\frac{n}{2}\right)^{n/2}$$

k

$\log_2$

$$O(n \log n)$$

$$\frac{n}{2} \log_2 \frac{n}{2}$$

Lower bound: How many comparisons will we need?

How many comparisons will any comparison-based sorting algorithm need to make?

There are n! different orderings for n elements
Each set of comparisons partitions the orderings into sets of “compatible” orderings

If $2^{\mathrm{k}} < n!$, by the pigeonhole principle, there are orderings that we can't distinguish with k comparisons. -> We need *k* such that $2^k \geq n!$, for a *rough estimate* we can use $n! \geq \left({}^{n}\!/_{2}\right)^{{}^{n}\!/_{2}}$
Therefore $k \geq {}^{n}\!/_{2} \log_2 {}^{n}\!/_{2}$

To obtain a better estimate we can use $n! \geq e \left({}^{n}\!/_{e}\right)^{n}$
which gives $k \geq 1.4\,(n-1)\ln n$

Expected Algorithm Runtime: Quicksort

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick first element e from S
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

Worst case running time is $O(n^2)$.

BUT what should we **expect** to happen?

best case: n log n

worst case: n^2

Expected Algorithm Runtime: Quicksort (->)

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick first element e from S
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

best case: n log n

average case?

worst case: n^2

Expected Algorithm Runtime: Quicksort (->)

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

"Randomised Algorithm"

best case: n log n

(~~average case~~)

expected runtime for worst input

worst case: n^2

Probability

the **relative frequency** of a probabilistic event

$$\frac{\textit{how many times the "experiment" was "successful"}}{\textit{how many times you conducted an "experiment"}} \quad \text{in expectation}$$

Probability

set of “successful” outcomes

$P(\omega \in A) = P(A)$

random event

the **relative frequency** of a probabilistic event

$$\frac{\textit{how many times the “experiment” was “successful”}}{\textit{how many times you conducted an “experiment”}}$$

in expectation

Probability

$P(\omega \in A) = \underline{P(A)}$

the **relative frequency** of a probabilistic event

$$\frac{\textit{how many times the “experiment” was “successful”}}{\textit{how many times you conducted an “experiment”}} \quad \text{in expectation}$$

if there is no reason that one event would happen more often than another

$$P(\omega) = {}^{1}\!/_{\textit{how many different events can happen}} = {}^{1}\!/_{|\Omega|}$$

Probability

sample space Ω, possible events $\boxed{2^\Omega}$, probability $P: 2^\Omega \to \mathbb{R}^+$

$P(\emptyset) = 0, P(\Omega) = 1, \forall A, B \subseteq \Omega: (A \cap B = \emptyset) \Rightarrow P(A \cup B) = P(A) + P(B)$

the **relative frequency** of a probabilistic event

Probability—fair dice—sampling “uniformly at random”

sample space Ω, possible events 2^{Ω}, probability $P: 2^{\Omega} \rightarrow \mathbb{R}^{+}$

$P(\emptyset) = 0, P(\Omega) = 1, \forall A, B \subseteq \Omega: (\, A \cap \mathrm{B} = \emptyset \,) \Rightarrow \; P(A \cup B) = P(A) + P(B)$

We call the **relative frequency** of a probabilistic event,

it’s probability e.g. $\underbrace{P(6) = P(5) = \; \ldots}$ $= \frac{1}{6}; P(\{5{,}6\}) = \frac{1}{3}$

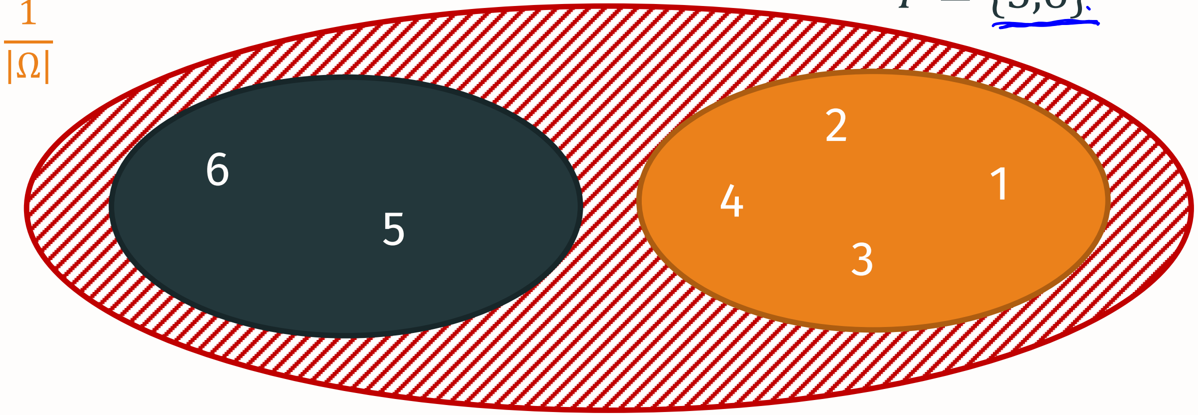
For fair dice (=uniform sampling)

$\forall \omega \in \Omega: P(\omega) = \dfrac{1}{|\Omega|}$

$P(T) = \dfrac{|T|}{|\Omega|}$

$\Omega = [\![6]\!]$

$T = \{5{,}6\}$



Random Variables & Expectation—linearity

Random variables assign values to outcomes of random experiments:

$$X\colon \Omega \to \mathbb{R}$$

The expected value of is a random variable's **probability-weighted** average:

$$\mathbb{E}(X) = \sum_{\omega \in \Omega} X(\omega) \cdot P(\omega)$$

Random Variables & Expectation

Random variables assign values to outcomes of random experiments:

$$X\colon \Omega \to \mathbb{R}$$

Examples

- number of eyes on a dice
- payoff after a game was won/lost
- number of times an event occurs in repeated experiments
- runtime of an algorithm

over input and algorithm

The expected value of is a random variable's **probability-weighted** average:

$$\mathbb{E}(X) = \sum_{\omega \in \Omega} X(\omega) \cdot P(\omega)$$

Examples

- average number of eyes thrown with a dice
- average payoff
- average number of times an event occurs in repeated experiments
- average runtime of an algorithm
- **worst-case expected runtime of an algorithm**

over input over alg

Expected Algorithm Runtime: Quicksort

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

Worst case running time is $O(n^2)$.

BUT what should we **expect** to happen?

Expected Algorithm Runtime: Quicksort

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

Worst case running time is $O(n^2)$.

BUT what should we **expect** to happen?

wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if i was compared to j

$$\boxed{X} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

Expected Algorithm Runtime: Quicksort

wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if i was compared to j

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

<u>**QuickSort**</u>**(S)**
input: set S of numbers
output: sorted list of elements from S
0: **if** $|S| \leq 1$ **then return** [e **for** e **in** S]
1: pick random element e from S (uniformly)
2: split S into subsets:

- set G of elements greater than e
- set L of elements less than e

3: **return** *QuickSort(L)* ⊕ [e] ⊕ *QuickSort(G)*

$$\mathbb{E}_X\left[\sum_i \sum_j X_{ij}\right] = \sum_i \sum_j \boxed{\mathbb{E}_{X_{ij}}[X_{ij}]}$$

Expected Algorithm Runtime: Quicksort

QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: **if** $|S| \leq 1$ **then return** [e **for** e **in** S]
1: pick random element e from S (uniformly)
2: split S into subsets:
- set G of elements greater than e
- set L of elements less than e

3: **return** *QuickSort(L)* ⊕ [e] ⊕ *QuickSort(G)*

wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if *i* was compared to *j*

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

now

$$P(X_{ij} = 1) = \text{?}$$

$P(\{5,7\} \mid \{5,6,7\})$

$P(X_{5,6} = 1) = 1$

$P(X_{5,7} = 1) = 2/3$

$P(X_{230,100} = 1) = \frac{2}{230-100+1}$

Expected Algorithm Runtime: Quicksort

QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: **if** $|S| \leq 1$ **then return** [e **for** e **in** S]
1: pick random element e from S (uniformly)
2: split S into subsets:

- set G of elements greater than e
- set L of elements less than e

3: **return** *QuickSort(L)* ⊕ [e] ⊕ *QuickSort(G)*

wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if i was compared to j

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

now

$$\boxed{P(X_{ij} = 1) = \frac{2}{j - i + 1}}$$

Expected Algorithm Runtime: Quicksort

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if i was compared to j

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

now

$$P(X_{ij} = 1) = \frac{2}{j - i + 1}$$

therefore

$$\mathbb{E}[X] \leq \sum_{i=1}^{n} 2 \cdot \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n - i + 1} \right)$$

$$+ \cdots \frac{1}{n}$$

Expected Algorithm Runtime: Quicksort

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if i was compared to j

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

now

$$P(X_{ij} = 1) = \frac{2}{j - i + 1}$$

therefore

$$\mathbb{E}[X] = \sum_{i=1}^{n} 2 \cdot \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n - i + 1} \right)$$

$$\leq 2 \cdot \sum_{i=1}^{n} \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right)$$

Excursion: Harmonic numbers

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} {}^{1}/_{i}$$

$n = 2^k$

$$H_{2^k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{2^k} = \sum_{i=1}^{2^k} {}^{1}/_{i}$$

Excursion: Harmonic numbers

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} {}^{1}/_{i}$$

$$-{}^{1}/_{2^k}$$

$$H_{2^k} = \underline{1} + \underbrace{\frac{1}{2} + \frac{1}{3}} + \underbrace{\cdots + \cancel{\frac{1}{2^k}}} = \sum_{i=1}^{2^k} {}^{1}/_{i} - \frac{1}{2^k}$$

$$\begin{aligned}
&\underline{1} \\
&+ \underbrace{\frac{1}{2} + \frac{1}{3}} \\
&+ \underbrace{\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}} \\
&\vdots \\
&+ \frac{1}{2^{k-1}} + \frac{1}{2^{k-1}+1} + \cdots + \frac{1}{2^k-1}
\end{aligned}$$

Excursion: Harmonic numbers

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} {}^{1}\!/_{i}$$

$$H_{2^k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{2^k} = \sum_{i=1}^{2^k} {}^{1}\!/_{i}$$

$$\begin{array}{rlll}
 & k & & k \\
 & \frac{1}{2} & \boxed{\le 1} & \le 1 & = 1 \\
\frac{1}{2} = & \frac{1}{4} + \frac{1}{4} & \le \frac{1}{2} + \frac{1}{3} & \le \frac{1}{2} + \frac{1}{2} & = 1 \\
\frac{1}{2} = & \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} & \le \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} & \le \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} & = 1 \\
 & & \cdots & & \\
\frac{1}{2} = & \frac{1}{2^k} + \frac{1}{2^k} + \cdots + \frac{1}{2^k} & \le \frac{1}{2^{k-1}} + \frac{1}{2^{k-1}+1} + \cdots + \frac{1}{2^k - 1} & \le \frac{1}{2^{k-1}} + \frac{1}{2^{k-1}} + \cdots + \frac{1}{2^{k-1}} & = 1
\end{array}
\left.\vphantom{\begin{array}{c}1\\1\\1\\1\\1\end{array}}\right\} k \text{ rows}$$

k rows

Excursion: Harmonic numbers

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} {}^{1}\!/_{i}$$

$$H_{2^k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \boxed{\frac{1}{2^k}} = \sum_{i=1}^{2^k} {}^{1}\!/_{i}$$

$$n = 2^k \quad \Rightarrow \quad \underline{\frac{1}{2}\log_2 n} \le H_n \le 1 + \log_2 n$$

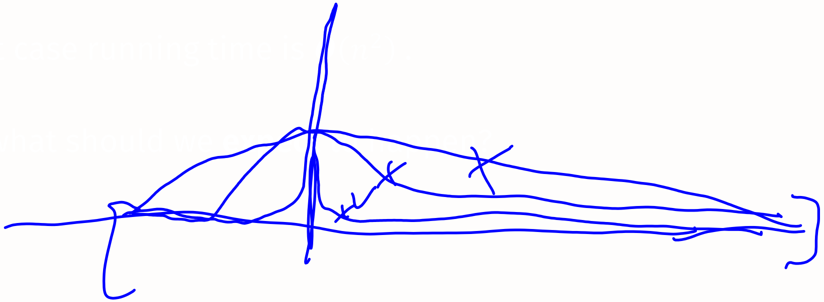
$k = \log_2 n \Leftrightarrow 2^k = n$

$$\begin{array}{lll}
\frac{1}{2} & \le 1 & \le 1 \\
\frac{1}{4} + \frac{1}{4} & \le \frac{1}{2} + \frac{1}{3} & \le \frac{1}{2} + \frac{1}{2} \\
\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} & \le \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} & \le \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} \\
 & \cdots & \\
\frac{1}{2^k} + \frac{1}{2^k} + \cdots + \frac{1}{2^k} & \le \frac{1}{2^{k-1}} + \frac{1}{2^{k-1}+1} + \cdots + \frac{1}{2^k-1} & \le \frac{1}{2^{k-1}} + \frac{1}{2^{k-1}} + \cdots + \frac{1}{2^{k-1}}
\end{array}$$

$$\frac{k}{2} \le \underline{H_{2^k}} - 2^{-k} \le k$$

Expected Algorithm Runtime: Quicksort

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```



wlog assume $S = [\![n]\!]$

random variable X_{ij} denotes if i was compared to j

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

now

$$P(X_{ij} = 1) = \frac{2}{j - i + 1}$$

therefore

$$\mathbb{E}[X] = \sum_{i=1}^{n} 2 \cdot \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n - i + 1}\right)$$

$$\leq 2 \cdot \sum_{i=1}^{n} \boxed{\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right)}$$

$$\leq \underline{2n \log_2 n}$$

Moore's Law

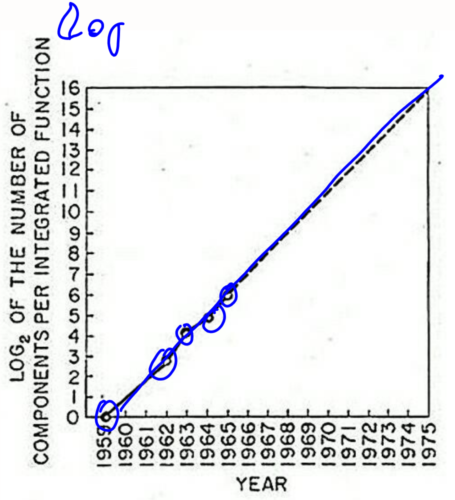
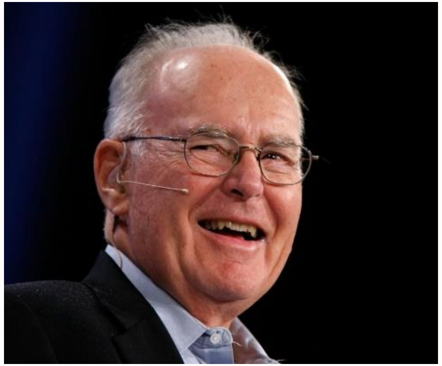


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.



[Gordon Moore](), 1965
["*Cramming more components onto integrated circuits*"]()

Moore’s Law

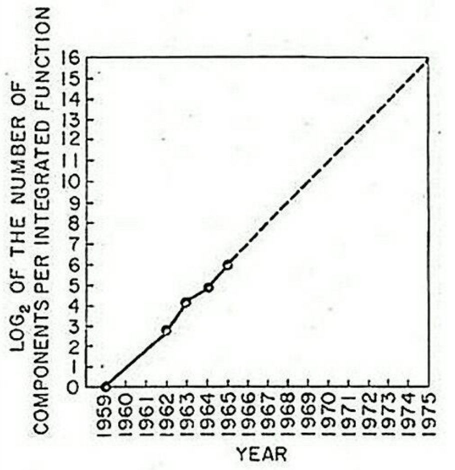
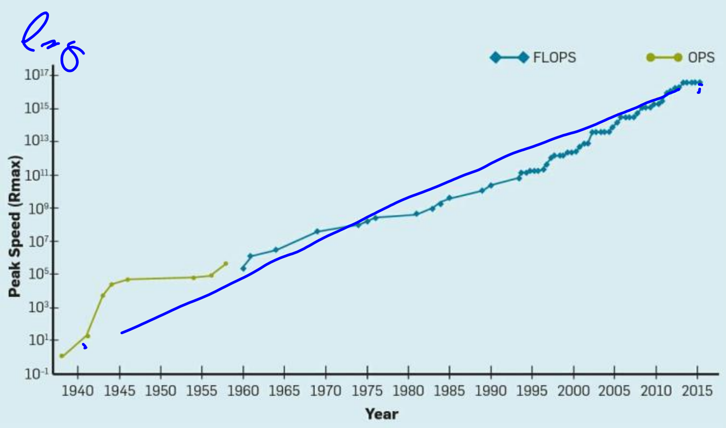


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.



Moore's Law

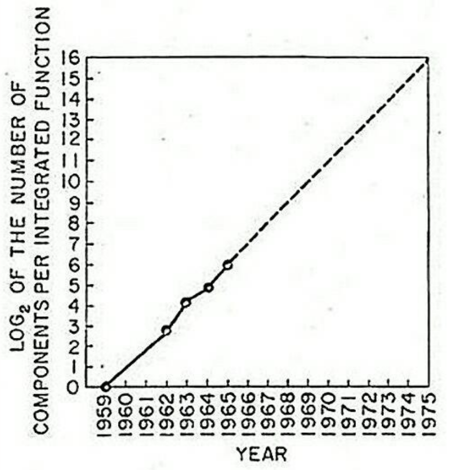
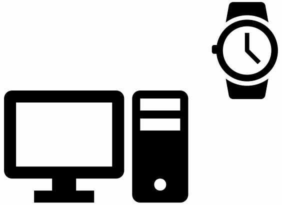
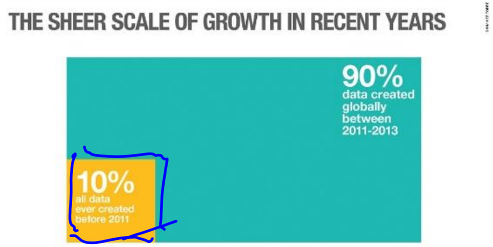
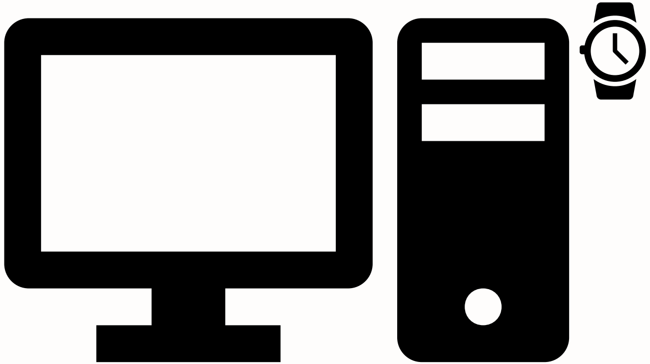


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.







Moore’s Law

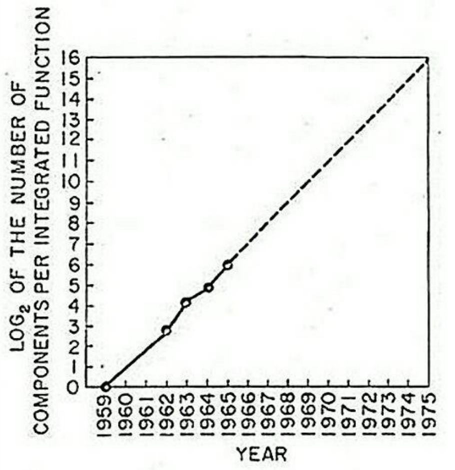
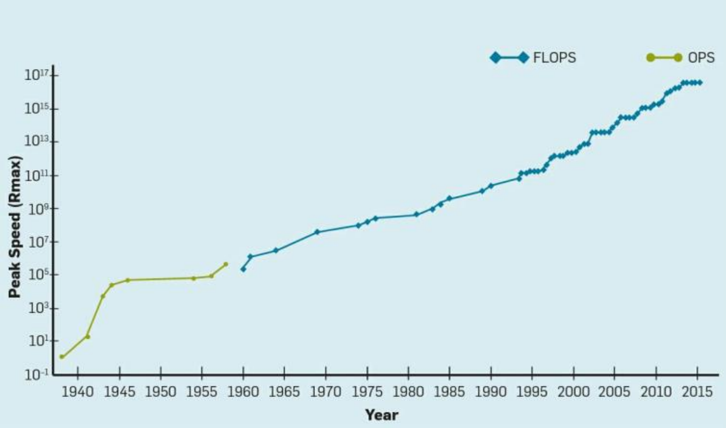
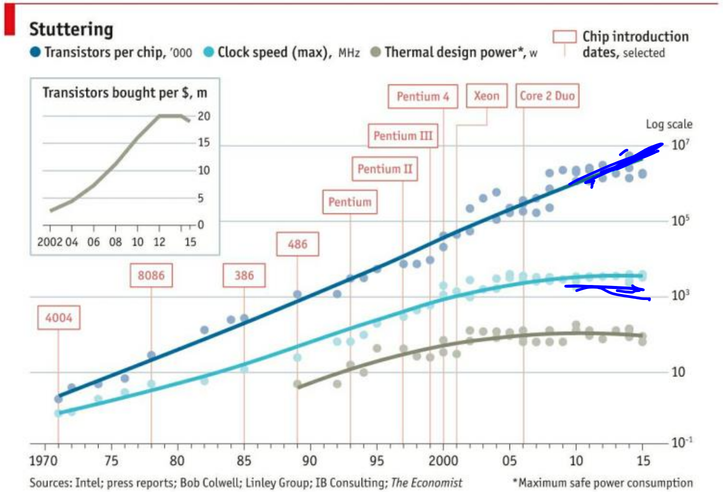
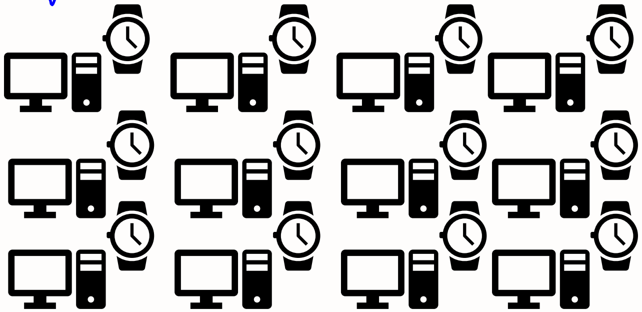


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.







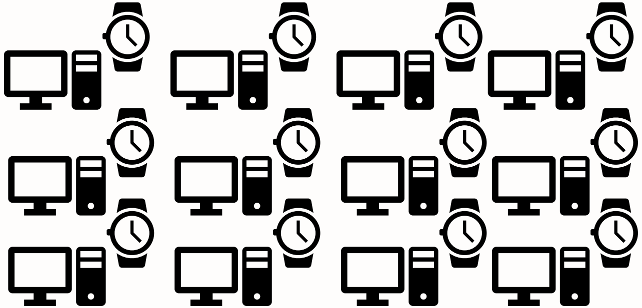
Moore's Law vs Nick's Class

[Nicholas John Pippenger]

[Stephen Cook]

The complexity class **NC** contains algorithms that

- need a **polylogarithmic** number of steps (in the worst case)



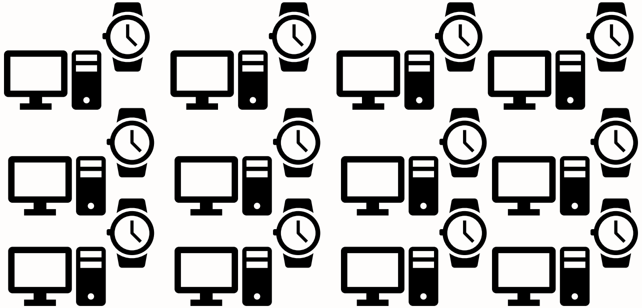
Moore's Law vs Nick's Class

[Nicholas John Pippenger]

[Stephen Cook]

The complexity class **NC** contains algorithms that

- are correct
- use a polynomial number of processors
- need a **polylogarithmic** number of steps (in the worst case)



PRAM – Parallel Random Access Machine

In each clock tick, each processor can read or write one memory location, or do some basic operation.

PRAM machines can have concurrent/exclusive reads and writes (CRCW, CREW, ERCW!, EREW).

Expected Algorithm Runtime: Quicksort (->)

```
QuickSort(S)
input: set S of numbers
output: sorted list of elements from S
0: if |S| ≤ 1 then return [e for e in S]
1: pick random element e from S (uniformly)
2: split S into subsets:
     • set G of elements greater than e
     • set L of elements less than e
3: return QuickSort(L) ⊕ [e] ⊕ QuickSort(G)
```

Worst case running time is $O(n^2)$.

BUT what should we **expect** to happen?

Parallel Quicksort (CRCW model)

<u>QuickSort</u>(S)
input: array s of numbers $s = (s_1, s_2, \ldots s_n)$ / processors $P_1, P_2, \ldots P_n$
ensure: array s is sorted
0: check *<u>in parallel</u>* **if** $s_1 \le s_2 \le \cdots s_n$ **then return**
1: pick random element e from S (uniformly)
2: *<u>in parallel</u>* **for all** i: **if** $s_i \le e$ **then** $b_i \leftarrow 0$ **else** $b_i \leftarrow 1$
3: compute *<u>in parallel</u>* prefix and postfix sums

$$r_i \leftarrow \sum_{j \le i} b_j \, , o_i \leftarrow \sum_{j \ge i} b_j \qquad (\forall i \in [\![n]\!])$$

4: **if** not ${}^{n}/_{4} \le r_n \le {}^{3n}/_{4}$ **then** goto 1
5: *<u>in parallel</u>* **for all** i: **if** $b_i = 0$ **then** $t_{r_i} \leftarrow s_i$ **else** $t_{o_i} \leftarrow s_i$
6: **return** *QuickSort*($t_{1:r_n}$) $\oplus$ *QuickSort*($t_{r_n+1:n}$)

time
log n

n processor

Worst case running time is $O(n^2)$.

BUT what should we expect to happen?

Parallel Quicksort (CRCW model)

QuickSort(S)

input: array s of numbers $s = (s_1, s_2, \dots s_n)$ / processors $P_1, P_2, \dots P_n$

ensure: array s is sorted

0: check *in parallel* **if** $s_1 \leq s_2 \leq \cdots s_n$ **then return**

1: pick random element e from S (uniformly)

2: *in parallel* **for all i: if** $s_i \leq e$ **then** $b_i \leftarrow 0$ **else** $b_i \leftarrow 1$

3: compute *in parallel* prefix and postfix sums

$$r_i \leftarrow \sum_{j \leq i} b_j \,,\, o_i \leftarrow \sum_{j \geq i} b_j \qquad (\forall i \in [\![n]\!])$$

4: **if** not $n/4 \leq r_n \leq 3n/4$ **then** goto 1

5: *in parallel* **for all i: if** $b_i = 0$ **then** $t_{r_i} \leftarrow s_i$ **else** $t_{o_i} \leftarrow s_i$

6: **return** *QuickSort*$(t_{1:r_n}) \oplus$ *QuickSort*$(t_{r_n+1:n})$

each *parallel instruction* can be performed in time log n

As before, we expect to need 2 ln n recursions

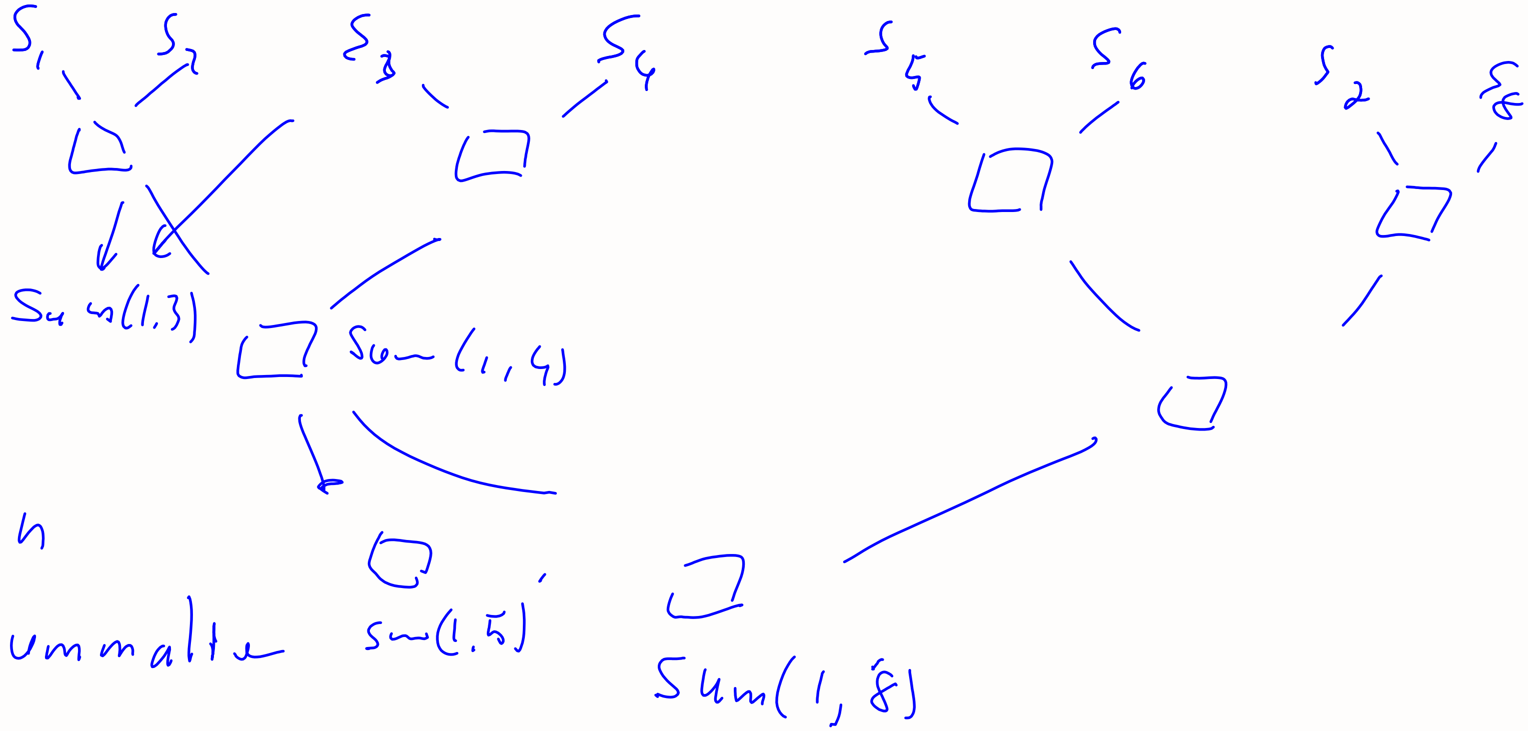
So the overall runtime is $\ln^2 n$ using n processors

Worst case running time is $O(n^2)$.

BUT what should we expect to happen?

O of n

summalte



Questions?