Departamento de Ciência de Computadores FCUP

Algoritmos (CC4010) 2019/20

Exame (21.01.2020) **Resolução** *duração: 3h (+30')*

1. [2.0] Relacione os conjuntos indicados usando o símbolo $\subset$, $\supset$, $\subseteq$, $\supseteq$, $=$, e $\neq$ que for **mais adequado**.

a) $\Omega(n^2+3n)$ $\boxed{\neq}$ $O(n^2)$ **b)** $\Theta(n^2)$ $\boxed{\subset}$ $\Omega(n\log n)$ **c)** $\Theta(\log_2 n)$ $\boxed{=}$ $\Theta(\log_{10} n)$

Justifique a alínea **(a)**, usando diretamente a definição matemática das ordens de grandeza.

Consideremos as funções $f_1(n) = n^3$ e $f_2(n) = n$.

$n^3 \in \Omega(n^2+3n)$ porque para $c = 1$ e $n_0 = 3$ tem-se $n^3 \geq c(n^2+3n)$, para todo $n \geq n_0$.

$n^3 \notin O(n^2)$ porque $\forall c' \in \mathbb{R}^+ \forall n'_0 \in \mathbb{Z}^+ \exists n \geq n'_0\ \ n^3 > c'n^2$. Basta tomar $n = \max(\lceil c' \rceil + 1, n'_0)$.

$n \in O(n^2)$ porque $n \leq n^2$, para todo $n \geq 1$.

$n \notin \Omega(n^2+3n)$ pois $\forall c'' \in \mathbb{R}^+ \forall n''_0 \in \mathbb{Z}^+ \exists n \geq n''_0\ \ n < c''(n^2+3n)$. Basta tomar $n = \max(\lceil 1/c'' - 3 \rceil + 1, n''_0)$.

Portanto, os conjuntos são distintos.

2. Seja $\mathcal{T}$ um conjunto de n tarefas que tem de realizar (se puder). A tarefa i teria início no instante t_i e duração d_i (ambos são inteiros positivos). Em cada instante só pode estar a realizar uma tarefa, mas pode começar uma nova tarefa no instante em que outra termina. Deve realizar **o número máximo de tarefas**.

a) [0.5] Prove que a estratégia que escolhe sempre a tarefa *com menor duração* que é compatível com as escolhidas anteriormente pelo mesmo algoritmo, pode produzir uma **solução não ótima**.

Consideremos três tarefas, com $t_1 = 1$ e $d_1 = 10$, $t_2 = 10$ e $d_2 = 3$, $t_3 = 11$ e $d_3 = 20$. O algoritmo escolheria apenas a tarefa 2, a qual é incompatível com as outras duas. A solução ótima é constituída pelas tarefas 1 e 3.

b) [1.0] Prove que a estratégia *"latest start first"*, que escolhe sempre a tarefa que *começa mais tarde* e é compatível com as escolhidas anteriormente pelo mesmo algoritmo, determina uma **solução ótima**.

Seja $S^\star$ uma solução ótima e S a obtida pelo algoritmo. Suponhamos que $S \neq S^\star$.

Comecemos **a analisar** $S^\star$ **e** S **do fim para o princípio** (ou seja, comparando as tarefas que em cada solução se iniciam mais tarde e sucessivamente). Nessa análise encontramos um primeiro par de tarefas distintas: seja $i^\star$ a tarefa em $S^\star$ e i a tarefa em S.

Quando o algoritmo (greedy) escolheu i, poderia ter escolhido $i^\star$ (pois as tarefas que escolheu antes são iguais às de $S^\star$). Como i começa depois ou ao mesmo tempo que $i^\star$, a tarefa i é compatível com todas as tarefas que começam antes de $i^\star$ em $S^\star$. Portanto, i pode substituir $i^\star$ em $S^\star$ e obtemos uma nova solução ótima, que difere menos de S. Procedendo de forma análoga com a nova solução ótima e S, acabamos por transformar a solução $S^\star$ (inicial) em S, o que demonstra a otimalidade de S (ou seja, o número de tarefas é o mesmo em ambas).

3. Suponha que se alterou a chave do elemento `k` numa heap de máximo `q`, com n elementos, e é necessário verificar (e, talvez repor) a propriedade de *heap*, usando `heapify(k,q)`.

a) [0.3] Em que consiste a "propriedade de *heap*" (para *heap de máximo*)?

Qualquer nó da heap é maior ou igual (segundo o critério de comparação) que os seus filhos, se existirem.

b) [1.7] Apresente em pseudocódigo a função `heapify(k,q)`, supondo definidas funções `troca(i,j,q)` e `compara(i,j,q)` para trocar os elementos `i` e `j`, entre si, e para comparar dois elementos (retornando 0 se forem iguais, um inteiro negativo se o primeiro for menor que o segundo, e positivo, caso contrário). Justifique a sua correção assumindo que as sub-árvores com raíz no filho esquerdo e direito do nó `k`, se existirem, satisfazem a propriedade de *heap*.

```
HEAPIFY(k, q)
    largest = k
    l = 2 * k
    r = 2 * k + 1
    if l ≤ q.size ∧ compara(l, largest, q) > 0 then
        largest = l
    if r ≤ q.size ∧ compara(r, largest, q) > 0 then
        largest = r
    if largest ≠ k then
        troca(largest, k, q)
        HEAPIFY(largest, q)
```

Se o nó k for maior ou igual que os seus filhos (se existirem), não é necessário prosseguir, pois a propriedade é satisfeita pelo nó k e os seus filhos e as as sub-árvores esquerda e direita são heaps de máximo também.

Se o nó k for menor que algum dos filhos então, quando **trocamos k com o filho maior**, garantimos que o outro filho (se existir) satisfaz a propriedade de heap relativamente ao novo pai, assim como a sub-árvore com raíz nesse filho. Apenas a sub-árvore que passou a ter o nó k como raíz poderá não satisfazer (ainda que as suas sub-árvores satisfaçam). Ao aplicar `heapify` a essa sub-árvore restabelecemos a propriedade.

c) [1.0] Na análise da complexidade assintótica de `heapify(1,q)`, usou-se $T(n) \leq T(2n/3) + c$, com c constante. Explique para quê e porquê. O que se conclui? Porque é que no **pior caso** $T(n) \in \Omega(\log_2 n)$.

Foi usada para majorar o tempo de execução de `heapify(1,q)` porque o número de comparações domina a complexidade da função e não excede o número de elementos da sub-árvore maior mais um (o qual se demonstrou que não pode exceder $(2/3)n$, sendo n o número de elementos da heap).

Conclui-se que $T(n) \in O(\log_2 n)$ pois a recorrência $T(n) = T(2n/3) + c$ tem solução $T(n) = \Theta(\log_2 n)$ pelo Master Theorem, caso 2, com $a = 1$, $b = 3/2$, $f(n) = c \in \Theta(1) = \Theta(n^0) = \Theta(n^{\log_{3/2} 1})$.

No pior caso, $T(n) \in \Omega(\log_2 n)$ porque se a raíz fosse o menor elemento, `heapify` teria de a deslocar até uma folha, fazendo trocas sucessivas ao longo de um caminho da raíz até tal folha, e a altura da árvore é $\lfloor \log_2 n \rfloor$.

4. [1.0] Explique sucintamente de que modo o algoritmo de Strassen, cuja complexidade é $\boxed{\Theta(n^{\log_2 7})}$, melhora a complexidade assintótica do produto de duas matrizes quadradas $n \times n$, face ao algoritmo trivial, cuja complexidade é $\boxed{\Theta(n^3)}$.

O algoritmo de Strassen baseia-se na estratégia *divide-and-conquer*. Explora propriedades algébricas para calcular o produto a partir de 7 produtos de matrizes $(n/2) \times (n/2)$, o que conduz à recorrência $T(n) = 7T(n/2) + \Theta(n^2)$, cuja solução é $T(n) \in \Theta(n^{\log_2 7})$, pelo Master Theorem, caso 1.

N.º Nome

5. Considere as funções MYPARTITION e SELECT assim definidas, sendo x é um *array* de n inteiros **distintos** $x[1], \ldots, x[n]$, e a e b inteiros tais que $1 \leq a \leq b \leq n$.

```
MYPARTITION(x, a, b)
1. z = x[a]
2. j = a
3. for i = a + 1 to b do
4.     if x[i] < z then
5.         Exchange x[i] with x[j + 1]
6.         j = j + 1
7. Exchange x[a] with x[j]
8. return j
```

```
SELECT(x, a, b, k)
1. if a > b ∨ b − a + 1 < k then
2.     return −1
3. t = MYPARTITION(x, a, b)
4. p = t − a + 1
5. if p = k then return x[t]
6. if p < k then
7.     return SELECT(x, t + 1, b, k − p)
8. return SELECT(x, a, t − 1, k)
```

a) [1.0] Indique o invariante de ciclo da função MYPARTITION, quando está a testar a condição de paragem para i fixo. O que se deduz sobre o resultado de MYPARTITION(x, a, b)?

Seja $v_a, v_{a+1}, \ldots, v_b$ o conteúdo do segmento $[a, b]$ de x à entrada da função. Para i fixo (com $a < i \leq b$), sabemos que foi analisada apenas a sequência $v_a, v_{a+1}, \ldots, v_{i-1}$, que z guarda v_a, mantendo-se $x[a] = v_a$. As posições de x desde $a + 1$ a j têm os elementos de $v_{a+1}, \ldots, v_{i-1}$ que são menores do que v_a, sendo $a \leq j < i$, e as posições desde $j + 1$ a $i - 1$ guardam os maiores (ou iguais). **O ciclo termina com** $i = b + 1$**.** Do invariante conclui-se que $v_{a+1}, \ldots, v_b$ foi analisada, e os valores menores do que v_a ficaram nas posições de $a + 1$ até j de x e os maiores (ou iguais) ficaram nas seguintes. Na linha 7, troca-se $x[a]$ (que é v_a) com $x[j]$. Se $j = a$ então essa operação não faz nada e, se $j \neq a$, o elemento com que se trocou faz parte do segmento $[a, b]$ e é menor do que v_a. Após a troca, os elementos menores do que v_a estão nas posições de a a $j - 1$ e os maiores (ou iguais) estão nas posições de $j + 1$ a b. A função retorna j que é a posição final do pivot v_a.

b) [1.0] Justifique a correção de SELECT(x, a, b, k), i.e., que obtém o k-ésimo menor elemento de $x[a], \ldots, x[b]$, se existir.

Se $a > b$ ou $b - a + 1 < k$, o k-ésimo menor elemento não existe e a função retorna -1 (que não é uma posição válida). Se $a \leq b$ e $b - a + 1 \geq k$, existe o k-ésimo menor elemento. De MYPARTITION concluimos que t fica com a posição do p-ésimo menor elemento de $v_a, \ldots, v_b$ (pois $t - a + 1$ é o número de elementos de $[a, t]$). Na linha 5, se $p = k$, retorna $x[t]$, o que é correto pois $x[t]$ seria o k-ésimo menor elemento. Se $p < k$ (linha 6), o valor procurado tem de estar no segmento $[t + 1, b]$ mas na chamada recursiva há que descontar os p elementos anteriores, para corrigir o número de ordem, o que é feito. Se $p > k$ (linha 8), o elemento está no segmento $[a, t - 1]$ e k não requer correção. Logo, a função está correta.

c) [1.5] Indique a complexidade temporal assintótica de SELECT(x, a, b, k) no melhor caso $\boxed{\Theta(1)}$, no pior caso $\boxed{\Theta((b - a + 1)^2)}$, e no caso médio $\boxed{\Theta(b - a + 1)}$ Para o caso médio, admita uma distribuição uniforme. Justifique a resposta que deu para o **pior caso**, começando por o descrever.

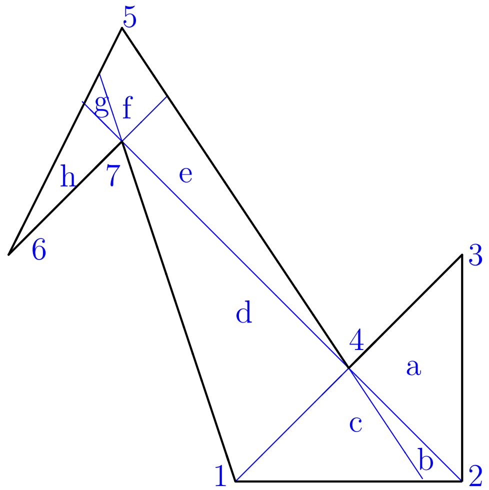
O pior caso acontece quando os elementos estão por ordem estritamente crescente e se pretende o máximo, i.e., o k-ésimo menor, com $k = b - a + 1$, sendo $1 \leq a \leq b \leq n$. Os elementos de x mantêm as posições iniciais em todas as chamadas de PARTITION e a recursão em SELECT, se $k > 1$, será sempre na linha 7, sobre $[a + 1, b], [a + 2, b], \ldots, [b - 1, b], [b, b]$, com $k = b - a, b - a - 1, \ldots, 2, 1$, o que conduz a $\sum_{i=1}^{b-a+1} i \in \Theta((b - a + 1)^2)$.

d) [1.0] Explique que vantagem teriam implementações baseadas em *quickselect* (com randomização) e *mediana das medianas de 5*, relativamente à indicada.

O tempo esperado para Quickselect é $\Theta(b - a + 1)$ (e no pior caso idêntico a SELECT) mas, por ser randomizado, nenhuma instância é sempre o pior caso. *Medians of 5* é determinístico e é $\Theta(b - a + 1)$ no pior caso, o que pode ser vantajoso.

6. [2.0] Considere o polígono com vértices $(2,0),(4,0),(4,2),(3,1),(1,4),(0,2),(1,3)$, numerados de 1 a 7. Apresente, **explicando**, um posicionamento de guardas que possa resultar da aplicação:

a) do algoritmo de Ghosh.

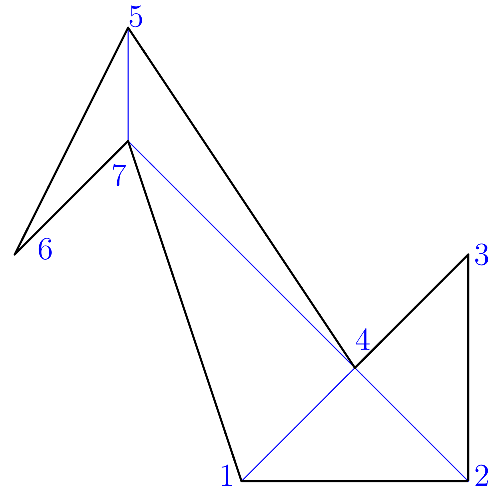


Reduz o problema a *minimum set cover (SC)*. Começa por obter a partição definida pelas regiões visíveis dos vértices.

$1:\{a,b,c,d,e,f\}$ $2:\{a,b,c,d\}$ $3:\{a,b,c\}$
$4:\{a,b,c,d,e,f,g\}$ $5:\{c,d,e,f,g,h\}$ $6:\{f,g,h\}$
$7:\{b,c,d,e,f,g,h\}$

Aplica o algoritmo greedy para *SC*: coloca um guarda em 4. Remove as peças que ficam cobertas e obtém $5:\{h\}, 6:\{h\}, 7:\{h\}$. Coloca um guarda em 5 (ou 6 ou 7).

b) da prova de Fisk para o teorema de Chvátal.



Efetua uma triangulação do polígono, que interpreta como um grafo. Determina uma coloração do grafo com três cores: preto - $\{3,1,5\}$, rosa - $\{2,7\}$, verde - $\{4,6\}$. Coloca os guardas nos nós com a cor **menos** frequente, por exemplo, nos vértices 2 e 7. Para concluir que $\lfloor n/3 \rfloor$ guardas são sempre suficientes, Fisk explora o facto de qualquer conjunto de guardas com a mesma cor cobrir todos os triângulos (e, portanto, o polígono fica coberto se se posicionar os guardas nos nós com a cor menos frequente, que não ocorrerá mais do que $\lfloor n/3 \rfloor$ vezes).

7. [1.5] Sejam X e Y duas sequências de carateres, com comprimento m e n, respetivamente. Considere o problema *Minimum Edit Distance*, aplicado a X e Y, sendo o custo da operação de inserção de 2, remoção 1 e substituição 2. Apresente a recorrência que define o custo da transformação de X_i em Y_j, sendo X_i e Y_j os prefixos de formados pelos i e j primeiros carateres, com $i \geq 0$ e $j \geq 0$. **Explique.**

Seja $C(i,j)$ o custo mínimo da transformação de X_i em Y_j. Pretendemos encontrar $C(m,n)$ e a recorrência é:

$$\begin{aligned} C(0,j) &= 2j, && \text{inserir os } j \text{ carateres de } Y_j\text{, para } 0 \leq j \leq n \\ C(i,0) &= i, && \text{remover os } i \text{ carateres de } X_i\text{, para } 0 \leq i \leq m \\ C(i,j) &= \min(C(i,j-1)+2, C(i-1,j)+1, C(i-1,j-1)+\alpha_{ij}), && \text{para } 1 \leq i \leq m, 1 \leq j \leq n \end{aligned}$$

com $\alpha_{ij} = 2$, se $X[i] \neq Y[j]$ (custo de substituição) e, caso contrário, $\alpha_{ij} = 0$.

No último caso, $C(i,j-1)+2$ seria o custo mínimo de transformar X_i em Y_{j-1} e inserir $Y[j]$, $C(i-1,j)+1$ seria o custo de transformar X_{i-1} em Y_j e remover $X[i]$, e $C(i-1,j-1)+\alpha_{ij}$ o custo de transformar X_{i-1} em Y_{j-1} e corrigir $(X[i], Y[j])$ se necessário.

8. [1.0] Explique sucintamente porque é que o problema de decisão associado a *minimum cardinality vertex cover* (em grafos não dirigidos) é da classe NP e porque é que o problema de otimização é da classe APX.

O problema de decisão é "Dado $G=(V,E)$ e $k \in \mathbb{Z}^+$, existe $C \subseteq V$ tal que qualquer aresta de E é incidente em C e $|C| \leq k$?". Pertence a NP porque, para qualquer instância cuja resposta seja "Sim", se for dado G, k e a prova C, existe um algoritmo que **verifica em tempo polinomial** que C satisfaz as condições (basta ver que $C \subseteq V$, que tem k ou menos elementos e que todas as arestas têm algum extremo em C, o que pode ser feito em tempo $\Theta(|V|+|E|)$). É da classe APX porque pertence a NPO e, nas aulas, foi estudado um algoritmo polinomial que produz uma aproximação de razão 2.

N.º | Nome

9. Considere uma *stack* suportada por um *array* V e uma variável top que designa o índice da primeira posição livre de V, com as operações usuais $\textsc{Push}(x)$ e $\textsc{Pop}()$ assim definidas:

$\textsc{Push}(x)$
  $V[top] = x$
  $top = top + 1$

$\textsc{Pop}()$
  $top = top - 1$
  return $V[top + 1]$

Suponha que inicialmente V tem M posições e considere a possibilidade de **a operação de $\textsc{Push}(x)$ ser substituída para aumentar dinamicamente a capacidade da stack** quando está cheia e é necessário inserir um novo valor. Nessa operação, começa por realocar espaço (um novo *array* com mais M posições do que o anterior, sendo M a constante inicial), copia os elementos que estavam na *stack* para o novo *array* e só depois insere o novo elemento. Considere um modelo de custos em que a inserção de um elemento e a remoção de um elemento têm custo 1 e o custo da expansão da estrutura de dados é igual ao número de elementos transferidos.

a) [1.0] Suponha que $M = 30$ e efetua uma sequência de 135 operações de $\textsc{Push}$, partindo da *stack vazia*. Apresente a expressão que define o custo total. Qual é o custo de uma operação de $\textsc{Push}$ no pior caso e no melhor caso? Qual é o custo amortizado de cada operação?

No melhor caso, o custo é 1. No pior caso é 121. O custo total é $135 + (30 + 60 + 90 + 120) = 435$.

O custo amortizado é $435/135$.

b) [1.0] Considere o caso geral (em que apenas se sabe que M é uma constante). Suponha que efetua uma sequência de kM operações de $\textsc{Push}$, sendo k inteiro positivo. Compare o custo amortizado de cada operação nesta abordagem com o custo amortizado se, em cada expansão, se *duplicasse* o tamanho do *array*.

O custo total é $kM + (M + 2M + \ldots + (k-1)M) = M\sum_{i=1}^{k} i = k(k+1)M/2$. O custo amortizado é $\frac{k(k+1)M/2}{kM} = \frac{k+1}{2}$ (ou seja, é $\Theta(k)$, e cresce se k cresce).

Se se optasse por duplicar o tamanho do array, o custo amortizado seria $O(1)$, pelo que seria preferível (foi visto nas aulas que nesse caso podemos definir o custo amortizado de cada $\textsc{Push}$ como 3).

10. No problema $\textsc{Bin Packing}$ são dados n itens com pesos $p_1, \ldots, p_n$, tais que $0 < p_i \leq 1$, para todo i, e há que os distribuir por latas de capacidade **unitária**, usando o menor número de latas possível.

No problema $\textsc{Partition}$ são dados n inteiros positivos $a_1, a_2, \ldots, a_n$, e há que decidir se existe uma partição $\{S, T\}$ do conjunto de índices $\{1, 2, \ldots, n\}$ tal que $\sum_{i \in S} a_i = \sum_{i \in T} a_i$. Sabe-se que $\textsc{Partition}$ é um problema **NP-completo**.

a) [0.2] Prove que as instâncias de $\textsc{Partition}$ com $a_j > \sum_{j \neq i} a_i$, para algum j, são trivialmente decidíveis.

Se $a_j > \sum_{j \neq i} a_i$, para algum j, a resposta a $\textsc{Partition}$ é "Não". Para resolver o problema, bastaria obter o máximo e a soma de todos os elementos e verificar se o dobro do máximo é maior do que soma, o que pode ser feito em $\Theta(n)$.

b) [1.0] Dada uma instância de $\textsc{Partition}$, com $a_j \leq \sum_{j \neq i} a_i$, para todo j, definimos uma instância de $\textsc{Bin Packing}$ com $p_j = 2a_j / \sum_{i=1}^{n} a_i$, para todo j. Justifique que se trata de uma redução polinomial que permite decidir $\textsc{Partition}$ em tempo polinomial se algum dos algoritmos seguintes existir e conclua que não podem existir a menos que P=NP: (i) um algoritmo polinomial que calcule uma solução ótima para $\textsc{Bin Packing}$; (ii) um algoritmo de aproximação polinomial de razão c para $\textsc{Bin Packing}$, com $c < 3/2$.

Como $a_j \leq \sum_{j \neq i} a_i$, para todos os j, então os pesos p_j satisfazem $0 < p_j \leq 1$. Por outro lado, $\sum_j p_j = 2$. Assim, para guardar $p_1, \ldots, p_n$ são necessárias pelo menos duas latas e **são necessárias exatamente duas latas, i.e., a solução ótima de BIN PACKING é 2 se e só se a resposta a PARTITION for "Sim"**.

Portanto, se calcularmos a solução ótima de BIN PACKING para a instância construída podemos dar a resposta a PARTITION. Assim, se existir um algoritmo polinomial para BIN PACKING, podiamos aplicá-lo à instância obtida e resolver PARTITION. Logo, a menos que P=NP, não existe um algoritmo polinomial para BIN PACKING.

Analogamente, se existir um algoritmo polinomial que produza uma α-aproximação para qualquer instância de BIN PACKING, com $\alpha < 3/2$, podiamos usá-lo para resolver PARTITION. Pois, se a resposta para PARTITION fosse "sim", a solução ótima da instância correspondente de BIN PACKING seria 2, e o algoritmo de aproximação teria de dar uma solução que usava menos do que $3/2 \times 2$ latas, i.e., menos de 3 latas. Sendo o número de latas inteiro, a solução (aproximada) usaria 2 latas (i.e., seria ótima). E, se a resposta a PARTITION for "Não", usaria pelo menos 3 latas. Portanto, a menos que P=NP, não existe um algoritmo polinomial que produza uma α-aproximação, com $\alpha < 3/2$, para BIN PACKING.

c) [0.3] Sabendo que estratégia "first fit decreasing" obtém em tempo polinomial uma aproximação de razão 3/2, conclua que BINPACKING é um problema APX-completo.

Ser APX-completo significa que pertence a APX e é APX-hard. O que se está a dizer é que BINPACKING pertence à classe APX e o que se concluiu acima foi que era APX-hard (pois, para $\alpha < 3/2$ não existe um algoritmo polinomial que produz uma α-aproximação para o problema, a menos que P=NP).

(Fim)

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 $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n)) = \sqrt{2\pi n}\left(\frac{n}{e}\right)^{\alpha_n}, \quad \text{with } 1/(12n+1) < \alpha_n < 1/(12n)$$

Some useful results:

$$\log(\prod_{k=1}^{n} a_k) = \sum_{k=1}^{n} \log a_k \qquad\qquad \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}, \quad \text{for } |x| < 1$$

If $(u_k)_k$ is an arithmetic progression (i.e., $u_{k+1} = r + u_k$, for some constant $r \neq 0$), then $\sum_{k=1}^{n} u_k = \frac{(u_1+u_n)n}{2}$.

If $(u_k)_k$ is a geometric progression (i.e., $u_{k+1} = ru_k$, for some constant $r \neq 1$), then $\sum_{k=1}^{n} u_k = \frac{u_{n+1}-u_1}{r-1}$.

If $f \geq 0$ is continuous and a monotonically increasing function, then

$$\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx$$